更新方法

意图

通过每次处理一帧的行为模拟一系列独立对象。

动机

玩家操作强大的女武神完成考验：从死亡巫王的栖骨之处偷走华丽的珠宝。 她尝试接近巫王华丽的地宫门口，然后遇到了……啥也没遇到。 没有诅咒雕像向她发射闪电，没有不死战士巡逻入口。 她直捣黄龙，拿走了珠宝。游戏结束。你赢了。

好吧，这可不行。

地宫需要守卫——一些英雄可以杀死的敌人。 首先，我们需要一个骷髅战士在门口前后移动巡逻。 如果无视任何关于游戏编程的知识， 让骷髅蹒跚着来回移动的最简单的代码大概是这样的：

> 如果巫王想表现得更加智慧，它应创造一些仍有脑子的东西。

```
while (true)
{
  // 向右巡逻
  for (double x = 0; x < 100; x++)
  {
    skeleton.setX(x);
  }

  // 向左巡逻
  for (double x = 100; x > 0; x--)
  {
    skeleton.setX(x);
  }
}
```

这里的问题，当然，是骷髅来回打转，可玩家永远看不到。 程序锁死在一个无限循环，那可不是有趣的游戏体验。 我们事实上想要的是骷髅每帧移动一步。

我们得移除这些循环，依赖外层游戏循环来迭代。 这保证了在卫士来回巡逻时，游戏能响应玩家的输入并进行渲染。如下：

> 当然，游戏循环是本书的另一个章节。

```
Entity skeleton;
bool patrollingLeft = false;
double x = 0;

// 游戏主循环
while (true)
{
  if (patrollingLeft)
  {
    x--;
    if (x == 0) patrollingLeft = false;
  }
  else
  {
    x++;
    if (x == 100) patrollingLeft = true;
  }

  skeleton.setX(x);

  // 处理用户输入并渲染游戏……
}
```

在这里前后两个版本展示了代码是如何变得复杂的。 左右巡逻需要两个简单的`for`循环。 通过指定哪个循环在执行，我们追踪了骷髅在移向哪个方向。 现在我们每帧跳出到外层的游戏循环，然后再跳回继续我们之前所做的，我们使用`patrollingLeft`显式地追踪了方向。

但或多或少这能行，所以我们继续。 一堆无脑的骨头不会对你的女武神提出太多挑战， 我们下一个添加的是魔法雕像。它们一直会向她发射闪电球，这样可让她保持移动。

继续我们的“用最简单的方式编码”的风格，我们得到了：

```
// 骷髅的变量……
Entity leftStatue;
Entity rightStatue;
int leftStatueFrames = 0;
int rightStatueFrames = 0;

// 游戏主循环：
while (true)
{
  // 骷髅的代码……

  if (++leftStatueFrames == 90)
  {
    leftStatueFrames = 0;
    leftStatue.shootLightning();
  }

  if (++rightStatueFrames == 80)
  {
    rightStatueFrames = 0;
    rightStatue.shootLightning();
  }

  // 处理用户输入，渲染游戏
}
```

你会发现这代码渐渐滑向失控。 变量数目不断增长，代码都在游戏循环中，每段代码处理一个特殊的游戏实体。 为了同时访问并运行它们，我们将它们的代码混杂在了一起。

> 一旦能用“混杂”一词描述你的架构，你就有麻烦了。

你也许已经猜到了修复这个所用的简单模式了：每个游戏实体应该封装它自己的行为。这保持了游戏循环的整洁，便于添加和移除实体。

为了做到这点需要抽象层，我们通过定义抽象的`update()`方法来完成。游戏循环管理对象的集合，但是不知道对象的具体类型。它只知道这些对象可以被更新。这样，每个对象的行为与游戏循环分离，与其他对象分离。

每一帧，游戏循环遍历集合，在每个对象上调用`update()`。这给了我们在每帧上更新一次行为的机会。在所有对象上每帧调用它，对象就能同时行动。

> 死抠细节的人会在这点上揪着我不放，是的，它们没有**真的同步**。当一个对象更新时，其他的都不在更新中。我们等会儿再说这点。

游戏循环维护动态的对象集合，所以从关卡添加和移除对象是很容易的——只需要将它们从集合中添加和移除。不必再用硬编码，我们甚至可以用数据文件构成这个关卡，那正是我们的关卡设计者需要的。

模式

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。

何时使用

如果游戏循环模式是切片面包，那么更新方法模式就是它的奶油。很多玩家交互的游戏实体都以这样或那样的方式实现了这个模式。如果游戏有太空陆战队，火龙，火星人，鬼魂或者运动员，很有可能它使用了这个模式。

但是如果游戏更加抽象，移动部分不太像活动的角色而更加像棋盘上的棋子，这个模式通常就不适用了。在棋类游戏中，你不需要同时模拟所有的部分，你可能也不需要告诉棋子每帧都更新它们自己。

> 你也许不需要每帧更新它们的**行为**，但即使是棋类游戏，你可能也需要每帧更新**动画**。这个设计模式也可以帮到你。

更新方法适应以下情况：

- 你的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他的大部分独立。
- 对象需要跟着时间进行模拟。

记住

这个模式很简单，所以没有太多值得发现的惊喜。当然，每行代码还是有利有弊。

将代码划分到一帧帧中会让它更复杂

当你比较前面两块代码时，第二块看上去更加复杂。两者都只是让骷髅守卫来回移动，但与此同时，第二块代码将控制权交给了游戏循环的一帧帧中。

几乎 这个改变是游戏循环处理用户输入，渲染等几乎必须要注意的事项，所以第一个例子不大实用。 但是很有必要记住，将你的行为切片会增加很高的复杂性。

> 我在这里说几乎是因为有时候鱼和熊掌可以兼得。 你可以直接为对象编码而不进行返回， 保持很多对象同时运行并与游戏循环保持协调。
>
> 你需要的是允许你同时拥有多个“线程”执行的系统。 如果对象的代码可以在执行中暂停和继续，而不是总得**返回**， 你可以用更加命令式的方式编码。
>
> 真实的线程太过重量级而不能这么做， 但如果你的语言支持轻量协同架构比如generators，coroutines或者fibers，那你也许可以使用它们。
>
> 字节码模式是另一个在应用层创建多个线程执行的方法。

当离开每帧时，你需要存储状态，以备将来继续。

在第一个示例代码中，我们不需要用任何变量表明守卫在向左还是向右移动。 这显式的依赖于哪块代码正在运行。

当我们将其变为一次一帧的形式，我们需要创建`patrollingLeft`变量来追踪行走的方向。 当从代码中返回时，就丢失了行走的方向，所以为了下帧继续，我们需要显式存储足够的信息。

状态模式通常可以在这里帮忙。 状态机在游戏中频繁出现的部分原因是（就像名字暗示的），它能在你离开时为你存储各种你需要的状态。

对象逐帧模拟，但并非真的同步

在这个模式中，游戏遍历对象集合，更新每一个对象。 在`update()`调用中，大多数对象都能够接触到游戏世界的其他部分， 包括现在正在更新的其他对象。这就意味着你更新对象的顺序至关重要。

如果对象更新列表中，A在B之前，当A更新时，它会看到B之前的状态。 但是当B更新时，由于A已经在这帧更新了，它会看见A的新状态。 哪怕按照玩家的视角，所有对象都是同时运转的，游戏的核心还是回合制的。 只是完整的“回合”只有一帧那么长。

> 如果，由于某些原因，你决定**不**让游戏按这样的顺序更新，你需要双缓冲模式。 那么AB更新的顺序就没有关系了，因为**双方**都会看对方之前那帧的状态。

当关注游戏逻辑时，这通常是件好事。 同时更新所有对象将把你带到一些不愉快的语义角落。 想象如果国际象棋中，黑白双方同时移动会发生什么。 双方都试图同时往同一个空格子中放置棋子。这怎么解决？

序列更新解决了这点——每次更新都让游戏世界从一个合法状态增量更新到下一个，不会出现引发歧义而需要协调的部分。

> 这对在线游戏也有用，因为你有了可以在网上发送的行动指令序列。

在更新时修改对象列表需小心

当你使用这个模式时，很多游戏行为在更新方法中纠缠在一起。 这些行为通常包括增加和删除可更新对象。

举个例子，假设骷髅守卫被杀死时掉落物品。 使用新对象，你通常可以将其增加到列表尾部，而不引起任何问题。 你会继续遍历这张链表，最终找到新的那个，然后也更新了它。

但这确实表明新对象在它产生的那帧就有机会活动，甚至有可能在玩家看到它之前。 如果你不想发生那种情况，简单的修复方法就是在游戏循环中缓存列表对象的数目，然后只更新那么多数目的对象就停止：

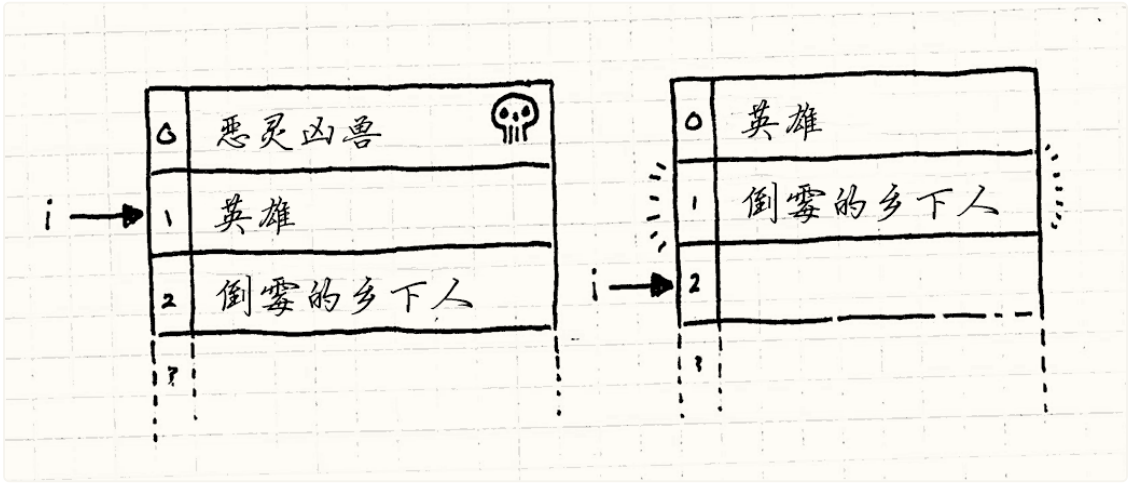
```
int numObjectsThisTurn = numObjects_;
for (int i = 0; i < numObjectsThisTurn; i++)
{
  objects_[i]->update();
}
```

这里，objects_是可更新游戏对象的数组，而numObjects_是数组的长度。 当添加新对象时，这个数组长度变量就增加。 在循环的一开始，我们在numObjectsThisTurn中存储数组的长度， 这样这帧的遍历循环会停在新添加的对象之前。

一个更麻烦的问题是在遍历时移除对象。 你击败了邪恶的野兽，现在它需要被移出对象列表。 如果它正好位于你当前更新对象之前，你会意外地跳过一个对象：

```
for (int i = 0; i < numObjects_; i++)
{
  objects_[i]->update();
}
```

这个简单的循环通过增加索引值来遍历每个对象。 下图的左侧展示了在我们更新英雄时，数组看上去是什么样的：



我们在更新她时，索引值i是1。 邪恶野兽被她杀了，因此需要从数组移除。 英雄移到了位置0，倒霉的乡下人移到了位置1。 在更新英雄之后，i增加到了2。 就像你在右图看到的，倒霉的乡下人被跳过了，没有更新。

一种简单的解决方案是在更新时**从后往前**遍历列表。 这种方式只会移动已经被更新的对象。

一种解决方案是小心地移除对象，任何对象被移除时，更新索引。 另一种是在遍历完列表后再移除对象。 将对象标为“死亡”，但是把它放在那里。 在更新时跳过任何死亡的对象。 然后，在完成遍历后，遍历列表并删除尸体。

如果在更新循环中有多个线程处理对象， 那么你可能更喜欢推迟任何修改，避免更新时同步线程的开销。

示例代码

这个模式太直观了，代码几乎只是在重复说明要点。 这不意味着这个模式没有用。它因为简单而有用：这是一个无需装饰的干净解决方案。

但是为了让事情更具体些，让我们看看一个基础的实现。 我们会从代表骷髅和雕像的`Entity`类开始：

```
class Entity
{
public:
  Entity()
  : x_(0), y_(0)
  {}

  virtual ~Entity() {}
  virtual void update() = 0;

  double x() const { return x_; }
  double y() const { return y_; }

  void setX(double x) { x_ = x; }
  void setY(double y) { y_ = y; }

private:
  double x_;
  double y_;
};
```

我在这里只呈现了我们后面所需东西的最小集合。 可以推断在真实代码中，会有很多图形和物理这样的其他东西。 上面这部分代码最重要的部分是它有抽象的`update()`方法。

游戏管理实体的集合。在我们的示例中，我会把它放在一个代表游戏世界的类中。

```
class World
{
public:
  World()
  : numEntities_(0)
  {}

  void gameLoop();

private:
  Entity* entities_[MAX_ENTITIES];
  int numEntities_;
};
```

> 在真实的世界程序中，你可能真的要使用集合类，我在这里使用数组来保持简单

现在，万事俱备，游戏通过每帧更新每个实体来实现模式：

```
void World::gameLoop()
{
  while (true)
  {
    // 处理用户输入……

    // 更新每个实体
    for (int i = 0; i < numEntities_; i++)
    {
      entities_[i]->update();
    }

    // 物理和渲染……
```

```
  }
}
```

> 正如其名，这是游戏循环模式的一个例子。

子类化实体？！

有很多读者刚刚起了鸡皮疙瘩，因为我在`Entity`主类中使用继承来定义不同的行为。 如果你在这里还没有看出问题，我会提供一些线索。

当游戏业界从6502汇编代码和VBLANKs转向面向对象的语言时， 开发者陷入了对软件架构的狂热之中。 其中之一就是使用继承。他们建立了遮天蔽日的高耸的拜占庭式对象层次。

最终证明这是个糟点子，没人可以不拆解它们来管理庞杂的对象层次。 哪怕在1994年的GoF都知道这点，并写道：

> 多用“对象组合”，而非“类继承”。

> 只在你我间聊聊，我认为这已经是一朝被蛇咬十年怕井绳了。 我通常避免使用它，但教条地不用和教条地使用一样糟。 你可以适度使用，不必完全禁用。

当游戏业界都明白了这一点，解决方案是使用组件模式。 使用它，`update()`是实体的组件而不是在`Entity`中。 这让你避开了为了定义和重用行为而创建实体所需的复杂类继承层次。相反，你只需混合和组装组件。

如果我真正在做游戏，我也许也会那么做。 但是这章不是关于组件的， 而是关于`update()`方法，最简单，最少牵连其他部分的介绍方法， 就是把更新方法放在`Entity`中然后创建一些子类。

> 组件模式在这里。

定义实体

好了，回到任务中。 我们原先的动机是定义巡逻的骷髅守卫和释放闪电的魔法雕像。 让我们从我们的骷髅朋友开始吧。 为了定义它的巡逻行为，我们定义恰当地实现了`update()`的新实体：

```
class Skeleton : public Entity
{
public:
  Skeleton()
  : patrollingLeft_(false)
  {}

  virtual void update()
  {
    if (patrollingLeft_)
    {
      setX(x() - 1);
      if (x() == 0) patrollingLeft_ = false;
    }
    else
    {
      setX(x() + 1);
      if (x() == 100) patrollingLeft_ = true;
    }
  }
```

```
private:
  bool patrollingLeft_;
};
```

如你所见，几乎就是从早先的游戏循环中剪切代码，然后粘贴到`Skeleton`的`update()`方法中。 唯一的小小不同是`patrollingLeft_`被定义为字段而不是本地变量。 通过这种方式，它的值在`update()`两次调用间保持不变。

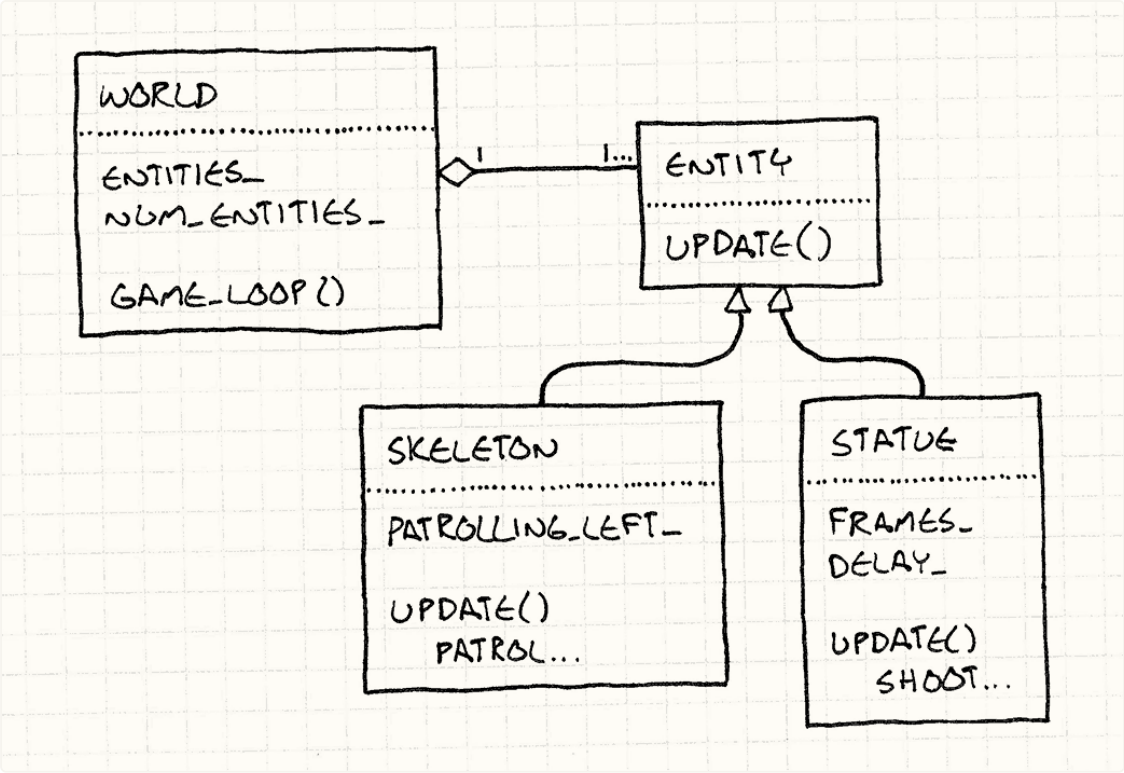
让我们对雕像如法炮制：

```
class Statue : public Entity
{
public:
  Statue(int delay)
  : frames_(0),
    delay_(delay)
  {}

  virtual void update()
  {
    if (++frames_ == delay_)
    {
      shootLightning();

      // 重置计时器
      frames_ = 0;
    }
  }

private:
  int frames_;
  int delay_;

  void shootLightning()
  {
    // 火光效果……
  }
};
```

又一次，大部分改动是将代码从游戏循环中移动到类中，然后重命名一些东西。 但是，在这个例子中，我们真的让代码库变简单了。 先前讨厌的命令式代码中，存在存储每个雕像的帧计数器和开火的速率的分散的本地变量。

现在那些都被移动到了`Statue`类中，你可以想创建多少就创建多少实例了， 每个实例都有它自己的小计时器。 这是这章背后的真实动机——现在为游戏世界增加新实体会更加简单， 因为每个实体都带来了它需要的全部东西。

这个模式让我们分离了游戏世界的构建和实现。 这同样能让我们灵活地使用分散的数据文件或关卡编辑器来构建游戏世界。



还有人关心UML吗？如果还有，那就是我们刚刚建的。

传递时间

这是模式的关键，但是我只对常用的部分进行了细化。 到目前为止，我们假设每次对`update()`的调用都推动游戏世界前进一个固定的时间。

我更喜欢那样，但是很多游戏使用可变时间步长。 在那种情况下，每次游戏循环推进的时间长度或长或短， 具体取决于它需要多长时间处理和渲染前一帧。

游戏循环一章讨论了更多关于固定和可变时间步长的优劣。

这意味着每次`update()`调用都需要知道虚拟的时钟转动了多少， 所以你经常可以看到传入消逝的时间。 举个例子，我们可以让骷髅卫士像这样处理变化的时间步长：

```
void Skeleton::update(double elapsed)
{
  if (patrollingLeft_)
  {
    x -= elapsed;
    if (x <= 0)
    {
      patrollingLeft_ = false;
      x = -x;
    }
  }
  else
  {
    x += elapsed;
    if (x >= 100)
    {
      patrollingLeft_ = true;
      x = 100 - (x - 100);
    }
  }
}
```

现在，骷髅卫士移动的距离随着消逝时间的增长而增长。 也可以看出，处理变化时间步长需要的额外复杂度。 如果一次需要更新的时间步长过长，骷髅卫士也许就超过了其巡逻的范围，因此需要小心的处理。

设计决策

在这样简单的模式中，没有太多的调控之处，但是这里仍有两个你需要决策的地方：

更新方法在哪个类中？

最明显和最重要的决策就是决定将`update()`放在哪个类中。

- 实体类中：

  如果你已经有实体类了，这是最简单的选项， 因为这不会带来额外的类。如果你需要的实体种类不多，这也许可行，但是业界已经逐渐远离这种做法了。

  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。 当你最终发现你想要用单一继承的方法重用代码时，你就卡住了。

- 组件类：

  如果你已经使用了组件模式，你知道这个该怎么做。 这让每个组件独立更新它自己。更新方法用了同样的方法解耦游戏中的实体，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。 状态模式可以这样做，你可以通过改变它委托的对象来改变它的行为。 类型对象模式也这样做了，这样你可以在同“种”实体间分享行为。

  如果你使用了这些模式，将`update()`放在委托类中是很自然的。 在那种情况下，也许主类中仍有`update()`方法，但是它不是虚方法，可以简单地委托给委托对象。就像这样：

  ```
  void Entity::update()
  {
    // 转发给状态对象
    state_->update();
  }
  ```

  这样做允许你改变委托对象来定义新行为。就像使用组件，这给了你无须定义全新的子类就能改变行为的灵活性。

如何处理隐藏对象？

游戏中的对象，不管什么原因，可能暂时无需更新。 它们可能是停用了，或者超出了屏幕，或者还没有解锁。 如果状态中的这种对象很多，每帧遍历它们却什么都不做是在浪费CPU循环。

一种方法是管理单独的“活动”对象集合，它存储真正需要更新的对象。 当一个对象停用时，从那个集合中移除它。当它启用时，再把它添加回来。 用这种方式，你只需要迭代那些真正需要更新的东西：

- 如果你使用单个包括了所有不活跃对象的集合：

- 浪费时间。对于不活跃对象，你要么检查一些“是否启用”的标识，要么调用一些啥都不做的方法。

> 检查对象启用与否然后跳过它，不但消耗了CPU循环，还报销了你的数据缓存。 CPU通过从RAM上读取数据到缓存上来优化读取。 这样做是基于刚刚读取内存之后的内存部分很可能等会儿也会被读取到这个假设。
>
> 当你跳过对象，你可能越过了缓存的尾部，强迫它从缓慢的主存中再取一块。

- 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，通常又需要一个巨大的集合。在那种情况下，这集合是多余的。 在速度比内存要求更高的时候（通常如此），这取舍仍是值得的。

另一个权衡后的选择是使用两个集合，除了活动对象集合的另一个集合只包含不活跃实体而不是全部实体。

  - 得保持集合同步。 当对象创建或完全销毁时（不是暂时停用），你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。

参见

- 这个模式，以及游戏循环模式和组件模式，是构建游戏引擎核心的三位一体。
- 当你关注在每帧中更新实体或组件的缓存性能时，数据局部性模式可以让它跑到更快。
- Unity框架在多个类中使用了这个模式，包括 `MonoBehaviour`。
- 微软的XNA平台在 `Game` 和 `GameComponent` 类中使用了这个模式。
- Quintus，一个JavaScript游戏引擎在它的主`Sprite`类中使用了这个模式。