# 组件模式



## 意图

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。

## 动机

让我们假设我们正在制作平台跳跃游戏。 意大利水管工已经有人做了，因此我们将出动丹麦面包师，Bjorn。 照理说，会有一个类来表示友好的糕点厨师，包含他在游戏中做的一切。

> 像这样的游戏创意导致了我是程序员而不是设计师。

由于玩家控制着他，这意味着需要读取控制器的输入然后转化为动作。 而且他当然需要与关卡进行互动，所以要引入物理和碰撞。 一旦这样做了，他就必须在屏幕上出现，所以要引入动画和渲染。 他可能还会播放一些声音。

等一下，这一切正在失控。软件体系结构101课程告诉我们，程序的不同领域应保持分离。 如果我们做一个文字处理器，处理打印的代码不应该受加载和保存文件的代码影响。 游戏和企业应用程序的领域不尽相同，但该规则仍然适用。

我们希望AI，物理，渲染，声音和其他领域域尽可能相互不了解， 但现在我们将所有这一切挤在一个类中。 我们已经看到了这条路通往何处：5000行的巨大代码文件，哪怕是你们团队中最勇敢的程序员也不敢打开。

这工作对能驯服他的少数人来说是有趣的，但对其他人而言是地狱。 这么大的类意味着，即使是看似微不足道的变化亦可有深远的影响。 很快，为类添加错误的速度会明显快于添加功能的速度。

### 一团乱麻

比起单纯的规模问题，更糟糕的是耦合。 在游戏中，所有不同的系统被绑成了一个巨大的代码球：

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
  playSound(HIT_FLOOR);
}
```

任何试图改变上面代码的程序员，都需要物理，图形和声音的相关知识，以确保没破坏什么。

> 这样的耦合在**任何**游戏中出现都是个问题，但是在使用并发的现代游戏中尤其糟糕。 在多核硬件上，让代码同时在多个线程上运行是至关重要的。 将游戏分割为多线程的一种通用方法是通过领域划分——在一个核上运行AI代码，在另一个上播放声音，在第三个上渲染，等等。
>
> 一旦你这么做了，在领域间保持解耦就是至关重要的，这是为了避免死锁或者其他噩梦般的并发问题。 如果某个函数从一个线程上调用`UpdateSounds()`方法，从另一个线程上调用`RenderGraphics()`方法，那它是在自找麻烦。

这两个问题互相混合；这个类涉及太多的域，每个程序员都得接触它， 但它又太过巨大，这就变成了一场噩梦。 如果变得够糟糕，程序员会黑入代码库的其他部分，仅仅为了躲开这个像毛球一样的`Bjorn`类。

## 快刀斩乱麻

我们可以像亚历山大大帝一样解决这个问题——快刀斩乱麻。 按领域将`Bjorn`类割成相互独立的部分。 例如，抽出所有处理用户输入的代码，将其移动到一个单独的`InputComponent`类。 `Bjorn`拥有这个部件的一个实例。我们将对`Bjorn`接触的每个领域重复这一过程。

当完成后，我们就将`Bjorn`大多数的东西都抽走了。 剩下的是一个薄壳包着所有的组件。通过将类划分为多个小类，我们已经解决了这个问题。但我们所完成的远不止如此。

## 宽松的结果

我们的组件类现在解耦了。 尽管`Bjorn`有`PhysicsComponent`和`GraphicsComponent`， 但这两部分都不知道对方的存在。 这意味着处理物理的人可以修改组件而不需要了解图形，反之亦然。

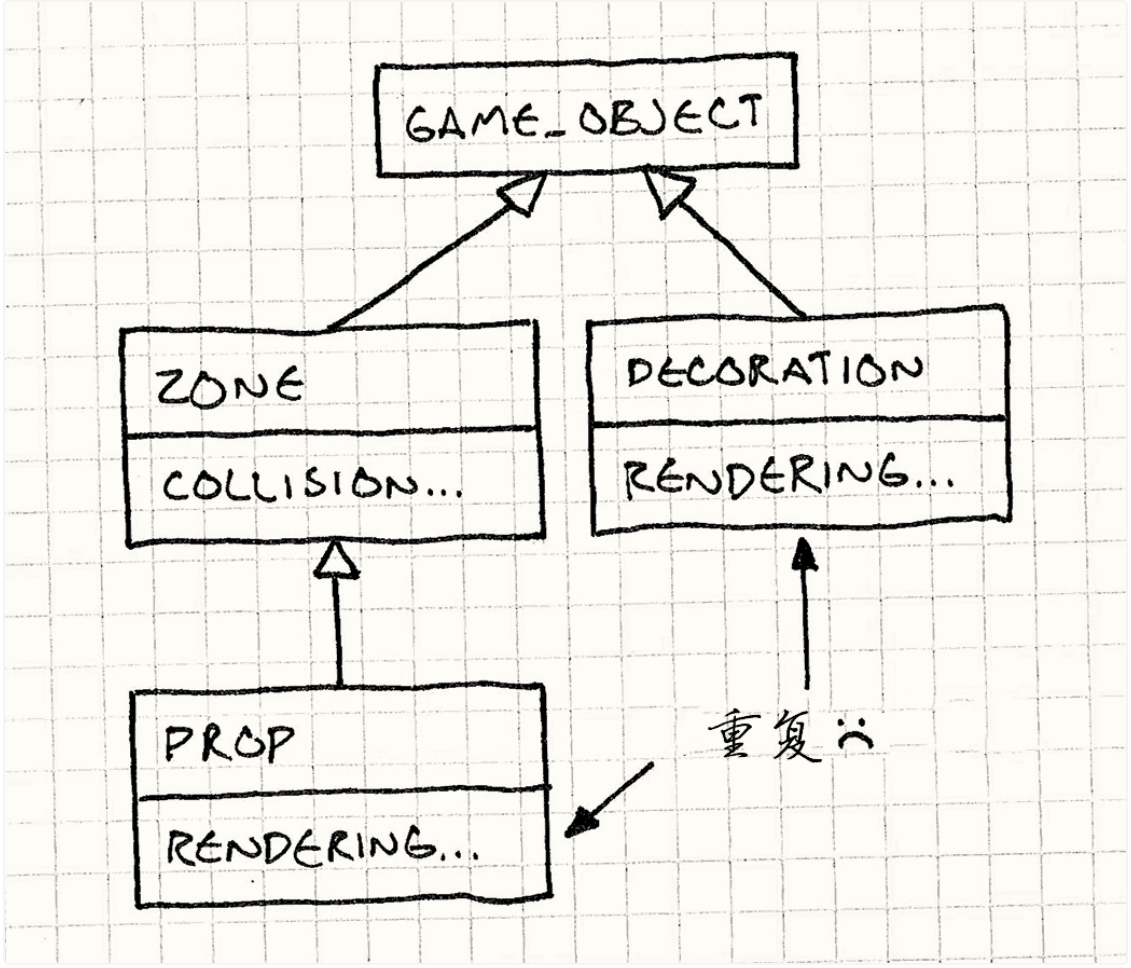
在实践中，这些部件之间需要有一些相互作用。 例如，AI组件可能需要告诉物理组件Bjorn试图去哪里。 然而，我们可以将这种交互限制在确实需要交互的组件之间， 而不是把它们围在同一个围栏里。

## 绑到一起

这种设计的另一特性是，组件现在是可复用的包。 到目前为止，我们专注于面包师，但是让我们考虑几个游戏世界中其他类型的对象。 装饰 是玩家看到但不能交互的事物：灌木，杂物等视觉细节。 道具 像装饰，但可以交互：箱，巨石，树木。 区域 与装饰相反——无形但可互动。 它们是很好的触发器，比如在Bjorn进入区域时触发过场动画。

> 当面向对象语言第一次接触这个场景时，继承是它箱子里最闪耀的工具。 它被认为是代码无限重用之锤，编程者常常挥舞着它。 然而我们痛苦地学到，事实上它是一把重锤。 继承有它的用处，但对简单的代码重用来说太过复杂。
>
> 相反，在今日软件设计的趋势是尽可能使用组件代替继承。 不是让两个类**继承**同一类来分享代码，而是让它们**拥有同一个类的实例。**

现在，考虑如果不用组件，我们将如何建立这些类的继承层次。第一遍可能是这样的：




我们有GameObject基类，包含位置和方向之类的通用部分。 Zone继承它，增加了碰撞检测。 同样，Decoration继承GameObject，并增加了渲染。 Prop继承Zone，因此它可以重用碰撞代码。 然而，Prop不能同时继承Decoration来重用渲染， 否则就会造成致命菱形结构。

> “致命菱形”发生在类继承了多个类，而这多个类中有两个继承同一基类时。 介绍它造成的痛苦超过了本书的范围，但它被说成“致命”是有原因的。

我们可以反过来让Prop继承Decoration，但随后不得不重复碰撞检测代码。 无论哪种方式，没有干净的办法重用碰撞和渲染代码而不诉诸多重继承。 唯一的其他选择是一切都继承GameObject， 但随后Zone会浪费内存在并不需要的渲染数据上， Decoration在物理效果上有同样的浪费。

现在，让我们尝试用组件。子类将彻底消失。 取而代之的是一个GameObject类和两个组件类：PhysicsComponent和GraphicsComponent。 装饰是个简单的GameObject，包含GraphicsComponent但没有PhysicsComponent。 区域与其恰好相反，而道具包含两种组件。 没有代码重复，没有多重继承，只有三个类，而不是四个。

> 可以拿饭店菜单打比方。如果每个实体是一个类，那就只能订套餐。 我们需要为每种**可能**的组合定义各自的类。 为了满足每位用户，我们需要十几种套餐。
>
> 组件是照单点菜——每位顾客都可以选他们想要的，菜单记录可选的菜式。

对对象而言，组件是即插即用的。 将不同的可重用部件插入对象，我们就能构建复杂且具有丰富行为的实体。 就像软件中的战神金刚。

# 模式

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。 实体被简化为组件的容器。

> “组件”，就像“对象”，在编程中意味任何东西也不意味任何东西。 正因如此，它被用来描述一些概念。 在商业软件中，“组件”设计模式描述通过网络解耦的服务。
>
> 我试图从游戏中找到无关这个设计模式的另一个名字，但“组件”看来是最常用的术语。 由于设计模式是记录已存的实践，我没有创建新术语的余地。 所以，跟着XNA，Delta3D和其他人的脚步，我称之为“组件”。

# 何时使用

组件通常在定义游戏实体的核心部分中使用，但它们在其他地方也有用。 这个模式应用在在如下情况中：

- 有一个涉及了多个领域的类，而你想保持这些领域互相隔离。
- 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。

# 记住

组件模式比简单地向类中添加代码增加了一点点复杂性。 每个概念上的“对象”要组成真正的对象需要实例化，初始化，然后正确地连接。 不同组件间沟通会有些困难，而控制它们如何使用内存就更加复杂。

对于大型代码库，为了解耦和重用而付出这样的复杂度是值得的。 但是在使用这种模式之前，保证你没有为了不存在的问题而“过度设计”。

使用组件的另一后果是，需要多一层跳转才能做要做的事。 拿到容器对象，获得相应的组件，然后 你才能做想做的事情。 在性能攸关的内部循环中，这种跳转也许会导致糟糕的性能。

> 这是硬币的两面。组件模式通常可以**增进**性能和缓存一致性。 组件让使用数据局部性模式的CPU更容易组织数据。

# 示例代码

我写这本书的最大挑战之一就是搞明白如何隔离各个模式。 许多设计模式包含了不属于这种模式的代码。 为了将提取模式的本质，我尽可能地消减代码， 但是在某种程度上，这就像是没有衣服还要说明如何整理衣柜。

说明组件模式尤其困难。 如果看不到它解耦的各个领域的代码，你就不能获得正确的体会， 因此我会多写一些有关于Bjorn的代码。 这个模式事实上只关于将组件变为类，但类中的代码可以帮助表明类是做什么用的。 它是伪代码——它调用了其他不存在的类——但这应该可以让你理解我们正在做什么。

## 单块类

为了清晰的看到这个模式是如何应用的， 我们先展示一个Bjorn类， 它包含了所有我们需要的事物，但是没有使用这个模式：

> 我应指出在代码中使用角色的名字总是个坏主意。市场部有在发售之前改名字的坏习惯。“焦点测试表明，在11岁到15岁之间的男性不喜欢‘Bjorn’，请改为‘Sven‘”。
>
> 这就是为什么很多软件项目使用内部代码名。 而且比起告诉人们你在完成“Photoshop的下一版本”，告诉他们你在完成“大电猫”更有趣。

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

  void update(World& world, Graphics& graphics);

private:
  static const int WALK_ACCELERATION = 1;

  int velocity_;
  int x_, y_;

  Volume volume_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

Bjorn有个每帧调用的update()方法。

```
void Bjorn::update(World& world, Graphics& graphics)
{
  // 根据用户输入修改英雄的速度
  switch (Controller::getJoystickDirection())
  {
    case DIR_LEFT:
      velocity_ -= WALK_ACCELERATION;
      break;

    case DIR_RIGHT:
      velocity_ += WALK_ACCELERATION;
      break;
  }

  // 根据速度修改位置
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // 绘制合适的图形
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0)
  {
    sprite = &spriteWalkLeft_;
  }
  else if (velocity_ > 0)
  {
    sprite = &spriteWalkRight_;
  }
```

```
  graphics.draw(*sprite, x_, y_);
}
```

它读取操纵杆以确定如何加速面包师。 然后，用物理引擎解析新位置。 最后，将Bjorn渲染至屏幕。

这里的示例实现平凡而简单。 没有重力，动画，或任何让人物有趣的其他细节。 即便如此，我们可以看到，已经出现了同时消耗多个程序员时间的函数，而它开始变得有点混乱。 想象增加到一千行，你就知道这会有多难受了。

## 分离领域

从一个领域开始，将Bjorn的代码去除一部分，归入分离的组件类。 我们从首个执行的领域开始：输入。 Bjorn做的头件事就是读取玩家的输入，然后基于此调整它的速度。 让我们将这部分逻辑移入一个分离的类：

```
class InputComponent
{
public:
  void update(Bjorn& bjorn)
  {
    switch (Controller::getJoystickDirection())
    {
      case DIR_LEFT:
        bjorn.velocity -= WALK_ACCELERATION;
        break;

      case DIR_RIGHT:
        bjorn.velocity += WALK_ACCELERATION;
        break;
    }
  }

private:
  static const int WALK_ACCELERATION = 1;
};
```

很简单吧。我们将Bjorn的update()的第一部分取出，放入这个类中。 对Bjorn的改变也很直接：

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);

    // 根据速度修改位置
    x += velocity;
    world.resolveCollision(volume_, x, y, velocity);

    // 绘制合适的图形
    Sprite* sprite = &spriteStand_;
    if (velocity < 0)
    {
      sprite = &spriteWalkLeft_;
    }
    else if (velocity > 0)
    {
      sprite = &spriteWalkRight_;
    }
```

```
    graphics.draw(*sprite, x, y);
  }

private:
  InputComponent input_;

  Volume volume_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

Bjorn现在拥有了一个InputComponent对象。 之前它在update()方法中直接处理用户输入，现在委托给组件：

```
input_.update(*this);
```

我们才刚开始，但已经摆脱了一些耦合——Bjorn主体现在已经与Controller无关了。这会派上用场的。

## 将剩下的分割出来

现在让我们对物理和图像代码继续这种剪切粘贴的工作。 这是我们新的PhysicsComponent：

```
class PhysicsComponent
{
public:
  void update(Bjorn& bjorn, World& world)
  {
    bjorn.x += bjorn.velocity;
    world.resolveCollision(volume_,
        bjorn.x, bjorn.y, bjorn.velocity);
  }

private:
  Volume volume_;
};
```

为了将物理行为移出Bjorn类，你可以看到我们也移出了数据：Volume对象已经是组件的一部分了。

最后，这是现在的渲染代码：

```
class GraphicsComponent
{
public:
  void update(Bjorn& bjorn, Graphics& graphics)
  {
    Sprite* sprite = &spriteStand_;
    if (bjorn.velocity < 0)
    {
      sprite = &spriteWalkLeft_;
    }
    else if (bjorn.velocity > 0)
    {
      sprite = &spriteWalkRight_;
    }

    graphics.draw(*sprite, bjorn.x, bjorn.y);
  }
```

```
private:
  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

我们几乎将所有的东西都移出来了，所以面包师还剩下什么？没什么了：

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Bjorn类现在基本上就做两件事：拥有定义它的组件，以及在不同域间分享的数据。 有两个原因导致位置和速度仍然在Bjorn的核心类中： 首先，它们是“泛领域”状态——几乎每个组件都需要使用它们， 所以我们想要提取它出来时，哪个组件应该拥有它们并不明确。

第二，也是更重要的一点，它给了我们无需让组件耦合就能沟通的简易方法。 让我们看看能不能利用这一点。

## 机器人Bjorn

到目前为止，我们将行为归入了不同的组件类，但还没将行为抽象出来。 Bjorn仍知道每个类的具体定义的行为。让我们改变这一点。

取出处理输入的部件，将其藏在接口之后，将InputComponent变为抽象基类。

```
class InputComponent
{
public:
  virtual ~InputComponent() {}
  virtual void update(Bjorn& bjorn) = 0;
};
```

然后，将现有的处理输入的代码取出，放进一个实现接口的类中。

```
class PlayerInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    switch (Controller::getJoystickDirection())
    {
      case DIR_LEFT:
        bjorn.velocity -= WALK_ACCELERATION;
        break;
```

```
      case DIR_RIGHT:
        bjorn.velocity += WALK_ACCELERATION;
        break;
    }
  }

private:
  static const int WALK_ACCELERATION = 1;
};
```

我们将Bjorn改为只拥有一个指向输入组件的指针，而不是拥有一个内联的实例。

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  Bjorn(InputComponent* input)
  : input_(input)
  {}

  void update(World& world, Graphics& graphics)
  {
    input_->update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

现在当我们实例化Bjorn，我们可以传入输入组件使用，就像下面这样：

```
Bjorn* bjorn = new Bjorn(new PlayerInputComponent());
```

这个实例可以是任何实现了抽象InputComponent接口的类型。 我们为此付出了代价——update()现在是虚方法调用了，这会慢一些。这一代价的回报是什么？

大多数的主机需要游戏支持“演示模式”。 如果玩家停在主菜单没有做任何事情，游戏就会自动开始运行，直到接入一个玩家。 这让屏幕上的主菜单看上去更有生机，同时也是销售商店里很好的展示。

隐藏在输入组件后的类帮我们实现了这点， 我们已经有了具体的PlayerInputComponent供玩游戏时使用。 现在让我们完成另一个：

```
class DemoInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    // 自动控制Bjorn的AI……
  }
};
```

当游戏进入演示模式，我们将Bjorn和一个新组件连接起来，而不像之前演示的那样构造它：

```
Bjorn* bjorn = new Bjorn(new DemoInputComponent());
```

现在，只需要更改组件，我们就有了为演示模式而设计的电脑控制的玩家。 我们可以重用所有Bjorn的代码——物理和图像都不知道这里有了变化。 也许我有些奇怪，但这就是每天能让我起床的事物。

> 那个，还有咖啡。热气腾腾的咖啡。

## 删掉Bjorn？

如果你看看现在的`Bjorn`类，你会意识到那里完全没有“Bjorn”——那只是个组件包。 事实上，它是个好候选人，能够作为每个游戏中的对象都能继承的“游戏对象”基类。 我们可以像弗兰肯斯坦一样，通过挑选拼装部件构建任何对象。

让我们将剩下的两个具体组件——物理和图像——像输入那样藏到接口之后。

```
class PhysicsComponent
{
public:
  virtual ~PhysicsComponent() {}
  virtual void update(GameObject& obj, World& world) = 0;
};

class GraphicsComponent
{
public:
  virtual ~GraphicsComponent() {}
  virtual void update(GameObject& obj, Graphics& graphics) = 0;
};
```

然后将`Bjorn`改为使用这些接口的通用`GameObject`类。

```
class GameObject
{
public:
  int velocity;
  int x, y;

  GameObject(InputComponent* input,
             PhysicsComponent* physics,
             GraphicsComponent* graphics)
  : input_(input),
    physics_(physics),
    graphics_(graphics)
  {}

  void update(World& world, Graphics& graphics)
  {
    input_->update(*this);
    physics_->update(*this, world);
    graphics_->update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent* physics_;
  GraphicsComponent* graphics_;
};
```

有些人走的更远。 不使用包含组件的`GameObject`，游戏实体只是一个ID，一个数字。 每个组件都知道它们连接的实体ID，然后管理分离的组件。

这些实体组件系统将组件发挥到了极致，让你向实体添加组件而无需通知实体。数据局部性一章有更多细节。

我们现有的具体类被重命名并实现这些接口：

```
class BjornPhysicsComponent : public PhysicsComponent
{
public:
  virtual void update(GameObject& obj, World& world)
  {
    // 物理代码……
  }
};

class BjornGraphicsComponent : public GraphicsComponent
{
public:
  virtual void update(GameObject& obj, Graphics& graphics)
  {
    // 图形代码……
  }
};
```

现在我们无需为Bjorn建立具体类，就能构建拥有所有Bjorn行为的对象。

```
GameObject* createBjorn()
{
  return new GameObject(new PlayerInputComponent(),
                        new BjornPhysicsComponent(),
                        new BjornGraphicsComponent());
}
```

这个createBjorn()函数当然就是经典的GoF工厂模式GoF的例子。

通过用不同组件实例化GameObject，我们可以构建游戏需要的任何对象。

# 设计决策

这章中你最需要回答的设计问题是“我需要什么样的组件？”回答取决于你游戏的需求和风格。 引擎越大越复杂，你就越想将组件划分得更细。

除此之外，还有几个更具体的选项要回答：

## 对象如何获取组件？

一旦将单块对象分割为多个分离的组件，就需要决定谁将它们拼到一起。

- 如果对象创建组件：
  - 这保证了对象总是能拿到需要的组件。 你永远不必担心某人忘记连接正确的组件然后破坏了整个游戏。容器类自己会处理这个问题。
  - 重新设置对象比较困难。 这个模式的强力特性之一就是只需重新组合组件就可以创建新的对象。 如果对象总是用硬编码的组件组装自己，我们就无法利用这个特性。
- 如果外部代码提供组件：
  - 对象更加灵活。 我们可以提供不同的组件，这样就能改变对象的行为。 通过共用组件，对象变成了组件容器，我们可以为不同目的一遍又一遍地重用它。

- 对象可以与具体的组件类型解耦。

如果我们允许外部代码提供组件，好处是也可以传递派生的组件类型。 这样，对象只知道组件接口而不知道组件的具体类型。这是一个很好的封装结构。

## 组件之间如何通信？

完美解耦的组件不需要考虑这个问题，但在真正的实践中行不通。 事实上组件属于同一对象暗示它们属于需要相互协同的更大整体的一部分。 这就意味着通信。

所以组件如何相互通信呢？ 这里有很多选项，但不像这本书中其他的“选项”，它们相互并不冲突——你可以在一个设计中支持多种方案。

- 通过修改容器对象的状态：
  - 保持了组件解耦。 当我们的`InputComponent`设置了Bjorn的速度，而后`PhysicsComponent`使用它， 这两个组件都不知道对方的存在。在它们的理解中，Bjorn的速度是被黑魔法改变的。
  - 需要将组件分享的任何数据存储在容器类中。 通常状态只在几个组件间共享。比如，动画组件和渲染组件需要共享图形专用的信息。 将信息存入容器类会让所有组件都获得这样的信息。

    更糟的是，如果我们为不同组件配置使用相同的容器类，最终会浪费内存存储不被任何对象组件需要的状态。 如果我们将渲染专用的数据放入容器对象中，任何隐形对象都会无益地消耗内存。
  - 这让组件的通信基于组件运行的顺序。 在同样的代码中，原先一整块的`update()`代码小心地排列这些操作。 玩家的输入修改了速度，速度被物理代码使用并修改位置，位置被渲染代码使用将Bjorn绘制到所在之处。 当我们将这些代码划入组件时，还是得小心翼翼地保持这种操作顺序。

    如果我们不那么做，就引入了微妙而难以追踪的漏洞。 比如，我们先更新图形组件，就错误地将Bjorn渲染在他上一帧而不是这一帧所处的位置上。 如果你考虑更多的组件和更多的代码，那你可以想象要避免这样的错误有多么困难了。

    > 这样被大量代码读写相同数据的共享状态很难保持正确。 这就是为什么学术界花时间研究完全函数式语言，比如Haskell，那里根本没有可变状态。
- 通过它们之间相互引用：

  这里的思路是组件有要交流的组件的引用，这样它们直接交流，无需通过容器类。

  假设我们想让Bjorn跳跃。图形代码想知道它需要用跳跃图像还是不用。 这可以通过询问物理引擎它当前是否在地上来确定。一种简单的方式是图形组件直接知道物理组件的存在：

```
class BjornGraphicsComponent
{
public:
  BjornGraphicsComponent(BjornPhysicsComponent* physics)
  : physics_(physics)
  {}

  void Update(GameObject& obj, Graphics& graphics)
  {
    Sprite* sprite;
    if (!physics_->isOnGround())
    {
      sprite = &spriteJump_;
```

```
    }
    else
    {
      // 现存的图形代码……
    }

    graphics.draw(*sprite, obj.x, obj.y);
  }

private:
  BjornPhysicsComponent* physics_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
  Sprite spriteJump_;
};
```

当构建Bjorn的GraphicsComponent时，我们给它相应的PhysicsComponent引用。

- 简单快捷。 通信是一个对象到另一个的直接方法调用。组件可以调用任一引用对象的方法。做什么都可以。
- 两个组件紧绑在一起。 这是做什么都可以带来的坏处。我们向使用整块类又退回了一步。 这比只用单一类好一点，至少我们现在只是把需要通信的类绑在一起。

- 通过发送消息：
  - 这是最复杂的选项。我们可以在容器类中建小小的消息系统，允许组件相互发送消息。

    下面是一种可能的实现。我们从每个组件都会实现的Component接口开始：

    ```
    class Component
    {
    public:
      virtual ~Component() {}
      virtual void receive(int message) = 0;
    };
    ```

    它有一个简单的receive()方法，每个需要接受消息的组件类都要实现它。 这里，我们使用一个int来定义消息，但更完整的消息实现应该可以附加数据。

    然后，向容器类添加发送消息的方法。

    ```
    class ContainerObject
    {
    public:
      void send(int message)
      {
        for (int i = 0; i < MAX_COMPONENTS; i++)
        {
          if (components_[i] != NULL)
          {
            components_[i]->receive(message);
          }
        }
      }

    private:
      static const int MAX_COMPONENTS = 10;
      Component* components_[MAX_COMPONENTS];
    };
    ```

现在，如果组件能够接触容器，它就能向容器发送消息，直接向所有的组件广播。（包括了原先发送消息的组件，小心别陷入无限的消息循环中！）这会造成一些结果：

> 如果你真的乐意，甚至可以将消息存储在**队列**中，晚些发送。 要知道更多，看看事件队列。

- 同级组件解耦。 通过父级容器对象，就像共享状态的方案一样，我们保证了组件之间仍然是解耦的。 使用了这套系统，组件之间唯一的耦合是它们发送的消息。

  > GoF称之为中介[GoF]模式——两个或更多的对象通过中介对象通信。 现在这种情况下，容器对象本身就是中介。

- 容器类很简单。 不像使用共享状态那样，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。

不出意料的，这里没有最好的答案。这些方法你最终可能都会使用一些。 共享状态对于每个对象都有的数据是很好用的——比如位置和大小。

有些不同领域仍然紧密相关。想想动画和渲染，输入和AI，或物理和粒子。 如果你有这样一对分离的组件，你会发现直接相互引用也许更加容易。

消息对于“不那么重要”的通信很有用。对物理组件发现事物碰撞后发送消息让音乐组件播放声音这种事情来说，发送后不管的特性是很有效的。

就像以前一样，我建议你从简单的开始，然后如果需要的话，加入其他的通信路径。

## 参见

- Unity核心架构中`GameObject`类完全根据这样的原则设计components。
- 开源的Delta3D引擎有`GameActor`基类通过`ActorComponent`实现了这种模式。
- 微软的XNA游戏框架有一个核心的`Game`类。它拥有一系列`GameComponent`对象。我们在游戏实体层使用组件，XNA在游戏主对象上实现了这种模式，但意图是一样的。
- 这种模式与GoF的策略模式[GoF]类似。 两种模式都是将对象的行为取出，划入单独的重述对象。 与对象模式不同的是，分离的策略模式通常是无状态的——它封装了算法，而没有数据。 它定义了对象如何行动，但没有定义对象是什么。

  组件更加重要。它们经常保存了对象的状态，这有助于确定其真正的身份。 但是，这条界限很模糊。有一些组件也许根本没有任何状态。 在这种情况下，你可以在不同的容器对象中使用相同的组件实例。这样看来，它的行为确实更像一种策略。