子类沙箱

游戏设计模式 / Behavioral Patterns

意图

用一系列由基类提供的操作定义子类中的行为。

动机

每个孩子都梦想过变成超级英雄，但是不幸的是，高能射线在地球上很短缺。 游戏是让你扮演超级英雄最简单的方法。 因为我们的游戏设计者从来没有学会说“不”，我们的超级英雄游戏中有成百上千种不同的超级能力可供选择。

我们的计划是创建一个`Superpower`基类。然后由它派生出各种超级能力的实现类。 我们在程序员队伍中分发设计文档，然后开始编程。 当我们完成时，我们就会有上百种超级能力类。

> 当你发现像这个例子一样有**很多**子类时，那通常意味着数据驱动的方式更好。 不再用**代码**定义不同的能力，用**数据**吧。
>
> 像类型对象，字节码，和解释器 GoF 模式都能帮忙。

我们想让玩家处于拥有无限可能的世界中。无论他们在孩童时想象过什么能力，我们都要在游戏中展现。 这就意味着这些超能力子类需要做任何事情： 播放声音，产生视觉刺激，与AI交互，创建和销毁其他游戏实体，与物理打交道。没有哪处代码是它们不会接触的。

假设我们让团队信马由缰地写超能力类。会发生什么？

- 会有很多冗余代码。 当超能力种类繁多，我们可以预期有很多重叠。 很多超能力都会用相同的方式产生视觉效果并播放声音。 当你坐下来看看，冷冻光线，热能光线，芥末酱光线都很相似。 如果人们实现这些的时候没有协同，那就会有很多冗余的代码和重复劳动。
- 游戏引擎中的每一部分都会与这些类耦合。 没有深入了解的话，任何人都能写出直接调用子系统的代码，但子系统从来没打算直接与超能力类绑定。 就算渲染系统被好好组织成多个层次，只有一个能被外部的图形引擎使用， 我们可以打赌，最终超能力代码会与每一个接触。
- 当外部代码需要改变时，一些随机超能力代码有很大几率会损坏。 一旦我们有了不同的超能力类绑定到游戏引擎的多个部分，改变那些部分必然影响超能力类。 这可不合理，因为图形，音频，UI程序员很可能不想也成为玩法程序员。

- 很难定义所有超能力遵守的不变量。 假设我们想保证超能力播放的所有音频都有正确的顺序和优先级。 如果我们的几百个类都直接调用音频引擎，就没什么好办法来完成这点。

我们要的是给每个实现超能力的玩法程序员一系列可使用的基本单元。 你想要播放声音？这是你的`playSound()`函数。 你想要粒子效果？这是你的`spawnParticles()`函数。我们保证了这些操作覆盖了你要做的事情，所以你不需要`#include`随机的头文件，干扰到代码库的其他部分。

我们实现的方法是通过定义这些操作为`Superpower`基类的**protected**方法。 将它们放在基类给了每个子类直接便捷的途径获取方法。 让它们成为**protected**（很可能不是虚方法）方法暗示了它们存在就是为了被子类调用。

一旦有了这些东西来使用，我们需要一个地方使用他们。 为了做到这点，我们定义沙箱方法，这是子类必须实现的抽象的**protected**方法。 有了这些，要实现一种新的能力，你需要：

1. 创建从`Superpower`继承的新类。
2. 重载沙箱方法`activate()`。
3. 通过调用`Superpower`提供的**protected**方法实现主体。

我们现在可以使用这些高层次的操作来解决冗余代码问题了。 当我们看到代码在多个子类间重复，我们总可以将其打包到`Superpower`中，作为它们都可以使用的新操作。

我们通过将耦合约束到一个地方解决了耦合问题。 `Superpower`最终与不同的系统耦合，但是继承它的几百个类不会。 相反，它们只耦合基类。 当游戏系统的某部分改变时，修改`Superpower`也许是必须的，但是众多的子类不需要修改。

这个模式带来浅层但是广泛的类层次。 你的继承链不深，但是有很多类与`Superpower`挂钩。 通过使用有很多直接子类的基类，我们在代码库中创造了一个支撑点。 我们投入到`Superpower`的时间和爱可以让游戏中众多类获益。

> 最近，你会发现很多人批评面向对象语言中的继承。 继承**是**有问题——在代码库中没有比父类子类之间的耦合更深的了——但我发现**扁平的**继承树比起**深的**继承树更好处理。

模式

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

子类沙箱模式是潜伏在代码库中简单常用的模式，哪怕是在游戏之外的地方亦有应用。 如果你有一个非虚的**protected**方法，你可能已经在用类似的东西了。 沙箱方法在以下情况适用：

- 你有一个能推导很多子类的基类。
- 基类可以提供子类需要的所有操作。
- 在子类中有行为重复，你想要更容易地在它们间分享代码。
- 你想要最小化子类和程序的其他部分的耦合。

记住

“继承”近来在很多编程圈子为人诟病，原因之一是基类趋向于增加越来越多的代码 这个模式特别容易染上这个毛病。

由于子类通过基类接触游戏的剩余部分，基类最后和子类需要的每个系统耦合。 当然，子类也紧密地与基类相绑定。这种蛛网耦合让你很难在不破坏什么的情况下改变基类——你得到了（脆弱的基类问题）brittle base class problem。

硬币的另一面是由于你耦合的大部分都被推到了基类，子类现在与世界的其他部分分离。理想的情况下，你大多数的行为都在子类中。这意味着你的代码库大部分是孤立的，很容易管理。

如果你发现这个模式正把你的基类变成一锅代码糊糊， 考虑将它提供的一些操作放入分离的类中， 这样基类可以分散它的责任。组件模式可以在这里帮上忙。

示例代码

因为这个模式太简单了，示例代码中没有太多东西。 这不是说它没用——这个模式关键在于“意图”，而不是它实现的复杂度。

我们从Superpower基类开始：

```
class Superpower
{
public:
  virtual ~Superpower() {}

protected:
  virtual void activate() = 0;

  void move(double x, double y, double z)
  {
    // 实现代码……
  }

  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void spawnParticles(ParticleType type, int count)
  {
    // 实现代码……
  }
};
```

activate()方法是沙箱方法。由于它是抽象虚函数，子类必须重载它。 这让那些需要创建子类的人知道要做哪些工作。

其他的protected函数move()，playSound()，和spawnParticles()都是提供的操作。 它们是子类在实现activate()时要调用的。

在这个例子中，我们没有实现提供的操作，但真正的游戏在那里有真正的代码。 那些代码中， Superpower 与游戏中其他部分的耦合——move() 也许调用物理代码，playSound()会与音频引擎交互，等等。 由于这都在基类的实现中，保证了耦合封闭在Superpower中。

好了，拿出我们的放射蜘蛛，创建个能力。像这样：

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    // 空中滑行
    playSound(SOUND_SPROING, 1.0f);
    spawnParticles(PARTICLE_DUST, 10);
    move(0, 0, 20);
  }
};
```

> 好吧，也许**跳跃**不是**超级能力**，但我在这里讲的是基础知识。

这种能力将超级英雄射向天空，播放合适的声音，扬起尘土。 如果所有的超能力都这样简单——只是声音，粒子效果，动作的组合——那么就根本不需要这个模式了。 相反，Superpower有内置的activate()能获取声音ID，粒子类型和运动的字段。 但是这只在所有能力运行方式相同，只在数据上不同时才可行。让我们精细一些：

```
class Superpower
{
protected:
  double getHeroX()
  {
    // 实现代码……
  }

  double getHeroY()
  {
    // 实现代码……
  }

  double getHeroZ()
  {
    // 实现代码……
  }

  // 退出之类的……
};
```

这里我们增加了些方法获取英雄的位置。我们的SkyLaunch现在可以使用它们了：

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    if (getHeroZ() == 0)
    {
      // 在地面上，冲向空中
      playSound(SOUND_SPROING, 1.0f);
      spawnParticles(PARTICLE_DUST, 10);
      move(0, 0, 20);
    }
    else if (getHeroZ() < 10.0f)
    {
      // 接近地面，再跳一次
      playSound(SOUND_SWOOP, 1.0f);
      move(0, 0, getHeroZ() + 20);
    }
    else
```

```
    {
      // 正在空中，跳劈攻击
      playSound(SOUND_DIVE, 0.7f);
      spawnParticles(PARTICLE_SPARKLES, 1);
      move(0, 0, -getHeroZ());
    }
  }
};
```

由于我们现在可以访问状态，沙箱方法可以做有用有趣的控制流了。 这还需要几个简单的`if`声明， 但你可以做任何你想做的东西。 使用包含任意代码的成熟沙箱方法，天高任鸟飞了。

> 早先，我建议以数据驱动的方式建立超能力。 这里是你可能**不**想那么做的原因之一。 如果你的行为复杂而使用命令式风格，它更难在数据中定义。

设计决策

如你所见，子类沙箱是一个“软”模式。它表述了一个基本思路，但是没有很多细节机制。这意味着每次使用都面临着一些有趣的选择。这里是一些需要思考的问题。

应该提供什么操作？

这是最大的问题。这深深影响了模式感觉上和实际上有多好。 在一个极端，基类几乎不提供任何操作。只有一个沙箱方法。 为了实现功能，总是需要调用基类外部的系统。如果你这样做，很难说你在使用这个模式。

另一个极端，基类提供了所有子类也许需要的操作。 子类只与基类耦合，不调用任何外部系统的东西。

> 具体来说，这意味着每个子类的源文件只需要`#include`它的基类头文件。

在这两个极端之间，操作由基类提供还是向外部直接调用有很大的操作余地。 你提供的操作越多，外部系统与子类耦合越少，但是与基类耦合越多。 从子类中移除了耦合是通过将耦合推给基类完成的。

如果你有一堆与外部系统耦合的子类的话，这很好。 通过将耦合移到提供的操作中，你将其移动到了一个地方：基类。但是你越这么做，基类就越大越难管理。

所以分界线在哪里？这里是一些首要原则：

- 如果提供的操作只被一个或几个子类使用，将操作加入基类获益不会太多。 你向基类添加了会影响所有事物的复杂性，但是只有少数几个类受益。

  让该操作与其他提供的操作保持一致或许有价值，但让使用操作的子类直接调用外部系统也许更简单明了。

- 当你调用游戏中其他地方的方法，如果方法没有修改状态就有更少的干扰。 它仍然制造耦合，但是这是“安全的”耦合，因为它没有破坏游戏中的任何东西。

  > “安全的”在这里打了引号是因为严格来说，接触数据也能造成问题。 如果你的游戏是多线程的，读取的数据可能正在被修改。如果你不小心，就会读入错误的数据。
  >
  > 另一个不愉快的情况是，如果你的游戏状态是严格确定性的（很多在线游戏为了保持玩家同步都是这样的）。 接触了游戏同步状态之外的东西会造成极糟的

不确定性漏洞。

另一方面，修改状态的调用会和代码库的其他方面紧密绑定，你需要三思。打包他们成基类提供的操作是个好的候选项。

- 如果操作只是增加了向外部系统的转发调用，那它就没增加太多价值。那种情况下，也许直接调用外部系统的方法更简单。

  但是，简单的转发也是有用的——那些方法接触了基类不想直接暴露给子类的状态。举个例子，假设Superpower提供这个：

  ```
  void playSound(SoundId sound, double volume)
  {
    soundEngine_.play(sound, volume);
  }
  ```

  它只是转发调用给Superpower中soundEngine_字段。 但是，好处是将字段封装在Superpower中，避免子类接触。

方法应该直接提供，还是包在对象中提供？

这个模式的挑战是基类中最终加入了很多方法。 你可以将一些方法移到其他类中来缓和。基类通过返回对象提供方法。

举个例子，为了让超能力播放声音，我们可以直接将它们加到Superpower中：

```
class Superpower
{
protected:
  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void stopSound(SoundId sound)
  {
    // 实现代码……
  }

  void setVolume(SoundId sound)
  {
    // 实现代码……
  }

  // 沙盒方法和其他操作……
};
```

但是如果Superpower已经很庞杂了，我们也许想要避免这样。 取而代之的是创建SoundPlayer类暴露该函数：

```
class SoundPlayer
{
  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void stopSound(SoundId sound)
  {
    // 实现代码……
  }

  void setVolume(SoundId sound)
```

```
  {
    // 实现代码……
  }
};
```

Superpower提供了对其的接触：

```
class Superpower
{
protected:
  SoundPlayer& getSoundPlayer()
  {
    return soundPlayer_;
  }

  // 沙箱方法和其他操作……

private:
  SoundPlayer soundPlayer_;
};
```

将提供的操作分流到辅助类可以为你做一些事情：

- 减少了基类中的方法。 在这里的例子中，将三个方法变成了一个简单的获取函数。
- 在辅助类中的代码通常更好管理。 像Superpower的核心基类，不管意图如何好，它被太多的类依赖而很难改变。 通过将函数移到耦合较少的次要类，代码变得更容易被使用而不破坏任何东西。
- 减少了基类和其他系统的耦合度。 当playSound()方法直接在Superpower时，基类与SoundId以及其他涉及的音频代码直接绑定。 将它移动到SoundPlayer中，减少了Superpower与SoundPlayer类的耦合，这就封装了它其他的依赖。

基类如何获得它需要的状态？

你的基类经常需要将对子类隐藏的数据封装起来。 在第一个例子中，Superpower类提供了spawnParticles()方法。 如果方法的实现需要一些粒子系统对象，怎么获得呢？

- 将它传给基类构造器：

  最简单的解决方案是让基类将其作为构造器变量：

  ```
  class Superpower
  {
  public:
    Superpower(ParticleSystem* particles)
    : particles_(particles)
    {}

    // 沙箱方法和其他操作……

  private:
    ParticleSystem* particles_;
  };
  ```

  这安全地保证了每个超能力在构造时能得到粒子系统。但让我们看看子类：

  ```
  class SkyLaunch : public Superpower
  {
  public:
    SkyLaunch(ParticleSystem* particles)
    : Superpower(particles)
  ```

```
  {}
};
```

我们在这儿看到了问题。每个子类都需要构造器调用基类构造器并传递变量。这让子类接触了我们不想要它知道的状态。

这也造成了维护的负担。如果我们后续向基类添加了状态，每个子类都需要修改并传递这个状态。

- 使用两阶初始化：

为了避免通过构造器传递所有东西，我们可以将初始化划分为两个部分。 构造器不接受任何参数，只是创建对象。然后，我们调用定义在基类的分离方法传入必要的数据：

```
Superpower* power = new SkyLaunch();
power->init(particles);
```

注意我们没有为SkyLaunch的构造器传入任何东西，它与Superpower中想要保持私有的任何东西都不耦合。 这种方法的问题在于，你要保证永远记得调用init()，如果忘了，你会获得处于半完成的，无法运行的超能力。

你可以将整个过程封装到一个函数中来修复这一点，就像这样：

```
Superpower* createSkyLaunch(ParticleSystem* particles)
{
  Superpower* power = new SkyLaunch();
  power->init(particles);
  return power;
}
```

> 使用一点像私有构造器和友类的技巧，你可以保证createSkylaunch()函数是唯一能够创建能力的函数。 这样，你不会忘记任何初始化步骤。

- 让状态静态化：

在先前的例子中，我们用粒子系统初始化每一个Superpower实例。 在每个能力都需要自己独特的状态时这是有意义的。但是如果粒子系统是单例，那么每个能力都会分享相同的状态。

如果是这样，我们可以让状态是基类私有而静态的。 游戏仍然要保证初始化状态，但是它只需要为整个游戏初始化Superpower类一遍，而不是为每个实例初始化一遍。

> 记住单例仍然有很多问题。你在很多对象中分享了状态（所有的Superpower实例）。 粒子系统被封装了，因此它不是全局**可见的**，这很好，但它们都访问同一对象，这让分析更加困难了。

```
class Superpower
{
public:
  static void init(ParticleSystem* particles)
  {
    particles_ = particles;
  }

  // 沙箱方法和其他操作……

private:
  static ParticleSystem* particles_;
};
```

注意这里的`init()`和`particles_`都是静态的。 只要游戏早先调用过一次`Superpower::init()`，每种能力都能接触粒子系统。 同时，可以调用正确的推导类构造器来自由创建`Superpower`实例。

更棒的是，现在`particles_`是静态变量， 我们不需要在每个`Superpower`中存储它，这样我们的类占据的内存更少了。

- 使用服务定位器：

  前一选项中，外部代码要在基类请求前压入基类需要的全部状态。 初始化的责任交给了周围的代码。另一选项是让基类拉取它需要的状态。 而做到这点的一种实现方法是使用服务定位器→模式：

```
class Superpower
{
protected:
  void spawnParticles(ParticleType type, int count)
  {
    ParticleSystem& particles = Locator::getParticles();
    particles.spawn(type, count);
  }

  // 沙箱方法和其他操作……
};
```

  这儿，`spawnParticles()`需要粒子系统，不是外部系统给它，而是它自己从服务定位器中拿了一个。

参见

- 当你使用更新模式→时，你的更新函数通常也是沙箱方法。
- 这个模式与模板方法GoF正相反。 两种模式中，都使用一系列受限操作实现方法。 使用子类沙箱时，方法在推导类中，受限操作在基类中。 使用模板方法时，基类 有方法，而受限操作在推导类中。
- 你也可以认为这个模式是外观GoF模式的变形。 外观模式将一系列不同系统藏在简化的API后。使用子类沙箱，基类起到了在子类前隐藏整个游戏引擎的作用。