服务定位器

提供服务的全局接入点，避免使用者和实现服务的具体类耦合。

动机

一些游戏中的对象或者系统几乎出现在程序库中的每一个角落。 很难找到游戏中的哪部分永远不需要内存分配，记录日志，或者随机数字。 像这样的东西可以被视为整个游戏都需要的服务。

我们考虑音频作为例子。 它不需要接触像内存分配这么底层的东西，但是仍然要接触一大堆游戏系统。 滚石撞击地面（物理）。 NPC狙击手开了一枪，射出子弹（AI）。 用户选择菜单项需要响一声确认（用户界面）。

每处都需要用像下面这样的东西调用音频系统：

```
// 使用静态类？
AudioSystem::playSound(VERY_LOUD_BANG);

// 还是使用单例？
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

尽管每种都能获得想要的结果，但是我们会绊倒在一些微妙的耦合上。 每个调用音频系统的游戏部分直接引用了具体的`AudioSystem`类，和访问它的机制——是静态类还是一个单例。 GoF

这些调用点，当然，需要耦合到某些东西上来播放声音， 但是直接接触到具体的音频实现，就好像给了一百个陌生人你家的地址，只是为了让他们在门口放一封信。 这不仅仅是隐私问题，在你搬家后，需要告诉每个人新地址是个更加痛苦的问题。

有个更好的解决办法：一本电话薄。 需要联系我们的人可以在上面查找并找到现在的地址。 当我们搬家时，我们通知电话公司。 他们更新电话薄，每个人都知道了新地址。 事实上，我们甚至无需给出真实的地址。 我们可以列一个转发信箱或者其他“代表”我们的东西。 通过让调用者查询电话薄找我们，我们获得了一个控制找我们的方法的方便地方。

这就是服务定位模式的简短介绍——它解耦了需要服务的代码和服务由谁提供（哪个具体的实现类）以及服务在哪里（我们如何获得它的实例）。

模式

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。

何时使用

当你需要让某物在程序的各处都能被访问时，你就是在找麻烦。 这是单例 GoF模式的主要问题，这个模式也没有什么不同。 我对何时使用服务定位器的最简单建议是：少用。

与其使用全局机制让某些代码接触到它，不如首先考虑将它传给代码。 这超简单，也明显保持了解耦，能覆盖你大部分的需求。

但是..... 有时候手动传入对象是不可能的或者会让代码难以阅读。 有些系统，比如日志或内存管理，不该是模块公开API的一部分。 传给渲染代码的参数应该与渲染相关，而不是与日志之类的相关。

同样，代表外设的系统通常只存在一个。 你的游戏可能只有一个音频设备或者显示设备。 这是周围环境的属性，所以将它传过十个函数让一个底层调用能够使用它会为代码增加不必要的复杂度。

如果是那样，这个模式可以帮忙。 就像我们将看到的那样，它是更加灵活、更加可配置的单例模式。 如果用得好，它能以很小的运行时开销，换取很大的灵活性。

> 相反，如果用得不好，它会带来单例模式的所有缺点以及更多的运行时开销。

记住

使用服务定位器的核心难点是它将依赖——在两块代码之间的一点耦合——推迟到运行时再连接。 这有了更大的灵活度，但是代价是更难在阅读代码时理解你依赖的是什么。

服务必须真的可定位

如果使用单例或者静态类，我们需要的实例不可能不可用。 调用代码保证了它就在那里。 但是由于这个模式是在定位服务，我们也许要处理失败的情况。 幸运的是，我们之后会介绍一种处理它的策略，保证我们在需要时总能获得某些服务。

服务不知道谁在定位它

由于定位器是全局可访问的，任何游戏中的代码都可以请求服务，然后使用它。 这就意味着服务必须在任何环境下正确工作。 举个例子，如果一个类只能在游戏循环的模拟部分使用，而不能在渲染部分使用，那它不适合作为服务——我们不能保证在正确的时间使用它。 所以，如果你的类只期望在特定上下文中使用，避免模式将它暴露给整个世界更安全。

示例代码

重回我们的音频系统问题，让我们通过服务定位器将代码暴露给代码库的剩余部分。

服务

我们从音频API开始。这是我们服务要暴露的接口：

```
class Audio
{
public:
  virtual ~Audio() {}
  virtual void playSound(int soundID) = 0;
  virtual void stopSound(int soundID) = 0;
  virtual void stopAllSounds() = 0;
};
```

当然，一个真实的音频引擎比这复杂得多，但这展示了基本的理念。 要点在于它是个没有实现绑定的抽象接口类。

服务提供者

只靠它自己，我们的音频接口不是很有用。 我们需要具体的实现。这本书不是关于如何为游戏主机写音频代码，所以你得想象这些函数中有实际的代码，了解原理就好：

```
class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID)
  {
    // 使用主机音频API播放声音……
  }

  virtual void stopSound(int soundID)
  {
    // 使用主机音频API停止声音……
  }

  virtual void stopAllSounds()
  {
    // 使用主机音频API停止所有声音……
  }
};
```

现在我们有接口和实现了。 剩下的部分是服务定位器——那个将两者绑在一起的类

一个简单的定位器

下面的实现是你可以定义的最简单的服务定位器：

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

  static void provide(Audio* service)
  {
    service_ = service;
  }

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

这里用的技术被称为**依赖注入**，一个简单思路的复杂行话表示。 假设你有一个类依赖另一个。 在例子中，是我们的Locator类需要Audio的实例。 通常，定位器负责构造实例。 依赖注入与之相反，它指外部代码负责向对象**注入**它需要的依赖。

静态函数getAudio()完成了定位工作。 我们可以从代码库的任何地方调用它，它会给我们一个Audio服务实例使用：

```
Audio *audio = Locator::getAudio();
audio->playSound(VERY_LOUD_BANG);
```

它“定位”的方式十分简单——依靠一些外部代码在任何东西使用服务前已注册了服务提供者。 当游戏开始时，它调用一些这样的代码：

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

这里需要注意的关键部分是调用playSound()的代码没有意识到任何具体的ConsoleAudio类； 它只知道抽象的Audio接口。 同样重要的是，定位器类没有与具体的服务提供者耦合。 代码中只有初始化代码唯一知道哪个具体类提供了服务。

这里有更高层次的解耦： Audio接口没有意识到它在通过服务定位器来接受访问。 据它所知，它只是常见的抽象基类。 这很有用，因为这意味着我们可以将这个模式应用到现有的类上，而那些类无需为此特殊设计。 这与单例[GoF]形成了对比，那个会影响“服务”类本身的设计。

一个空服务

我们现在的实现很简单，而且也很灵活。 但是它有巨大的缺点：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。

> 我有时听说这被称为“时序耦合”——两块分离的代码必须以正确的顺序调用，才能让程序正确运行。 有状态的软件某种程度上都有这种情况，但是就像其他耦合一样，减少时序耦合让代码库更容易管理。

幸运的是，还有一种设计模式叫做“空对象”，我们可用它处理这个。 基本思路是在我们没能找到服务或者程序没以正确的顺序调用时，不返回NULL， 而是返回一个特定的，实现了请求对象一样接口的对象。 它的实现什么也不做，但是它保证调用服务的代码能获取到对象，保证代码就像收到了“真的”服务对象一样安全运行。

为了使用它，我们定义另一个“空”服务提供者：

```
class NullAudio: public Audio
{
public:
  virtual void playSound(int soundID) { /* 什么也不做 */ }
  virtual void stopSound(int soundID) { /* 什么也不做 */ }
  virtual void stopAllSounds()        { /* 什么也不做 */ }
};
```

就像你看到的那样，它实现了服务接口，但是没有干任何实事。 现在，我们将服务定位器改成这样：

```
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_;
};
```

> 你也许注意到我们用引用而非指针返回服务。 由于C++中的引用（理论上）永远不是NULL，返回引用是提示用户：总可以期待获得一个合法的对象。
>
> 另一件值得注意的事是我们在provide()而不是访问者中检查NULL。 那需要我们早早调用initialize()，保证定位器可以正确找到默认的空服务提供者。 作为回报，它将分支移出了getAudio()，这在每次使用服务时节约了检查开销。

调用代码永远不知道“真正的”服务没找到，也不必担心处理NULL。 这保证了它永远能获得有效的对象。

这对故意找不到服务也很有用。 如果我们想暂时停用系统，现在有更简单的方式来实现这点了： 很简单，不要在定位器中注册服务，定位器会默认使用空服务提供器。

> 在开发中能关闭音频是很便利的。它释放了一些内存和CPU循环。 更重要的是，当你使用debugger时正好爆发巨响，它能防止你的鼓膜爆裂。 没有什么东西比二十毫秒的最高音量尖叫循环更能让你血液逆流的了。

日志装饰器

现在我们的系统非常强健了，让我们讨论这个模式允许的另一个好处——装饰服务。 我会举例说明。

在开发过程中，记录有趣事情发生的小小日志系统可助你查出游戏引擎正处于何种状态。 如果你在处理AI，你要知道哪个实体改变了AI状态。 如果你是音频程序员，你也许想记录每个播放的声音，这样你可以检查它们是否是以正确的顺序触发。

通常的解决方案是向代码中丢些对log()函数的调用。 不幸的是，这是用一个问题取代了另一个——现在我们有太多日志了。 AI程序员不关心声音在什么时候播放，声音程序员也不在乎AI状态转换，但是现在都得在对方的日志中跋涉。

理念上，我们应该可以选择性地为关心的事物启动日志，而游戏成品中，不应该有任何日志。 如果将不同的系统条件日志改写为服务，那么我们就可以用装饰器[GoF]模式。 让我们定义另一个音频服务提供者的实现：

```
class LoggedAudio : public Audio
{
public:
  LoggedAudio(Audio &wrapped)
  : wrapped_(wrapped)
  {}

  virtual void playSound(int soundID)
  {
```

```
    log("play sound");
    wrapped_.playSound(soundID);
  }

  virtual void stopSound(int soundID)
  {
    log("stop sound");
    wrapped_.stopSound(soundID);
  }

  virtual void stopAllSounds()
  {
    log("stop all sounds");
    wrapped_.stopAllSounds();
  }

private:
  void log(const char* message)
  {
    // 记录日志的代码……
  }

  Audio &wrapped_;
};
```

如你所见，它包装了另一个音频提供者，暴露同样的接口。 它将实际的音频行为转发给内部的提供者，但它也同时记录每个音频调用。 如果程序员需要启动音频日志，他们可以这样调用：

```
void enableAudioLogging()
{
  // 装饰现有的服务
  Audio *service = new LoggedAudio(Locator::getAudio());

  // 将它换进来
  Locator::provide(service);
}
```

现在，对音频服务的任何调用在运行前都会记录下去。 同时，当然，它和我们的空服务也能很好地相处，你能启用音频，也能继续记录音频被启用时将会播放的声音。

设计决策

我们讨论了一种典型的实现，但是对核心问题的不同回答有着不同的实现方式：

服务是如何被定位的？

- 外部代码注册：

  这是样例代码中定位服务使用的机制，这也是我在游戏中最常见的设计方式：

  - 简单快捷。 `getAudio()`函数简单地返回指针。这通常会被编译器内联，所以我们几乎没有付出性能损失就获得了很好的抽象层。
  - 可以控制如何构建提供者。 想想一个接触游戏控制器的服务。我们使用两个具体的提供者：一个是给常规游戏，另一个给在线游戏。 在线游戏跨过网络提供控制器的输入，这样，对游戏的其他部分，远程玩家好像是在使用本地控制器。

    为了能正常工作，在线的服务提供者需要知道其他远程玩家的IP。 如果定位器本身构建对象，它怎么知道传进来什么？ `Locator`类对在线的情况一无所知，更不用说其他用户的IP地址了。

外部注册的提供者闪避了这个问题。定位器不再构造类，游戏的网络代码实例化特定的在线服务提供器， 传给它需要的IP地址。然后把服务提供给定位器，而定位器只知道服务的抽象接口。

- 可以在游戏运行时改变服务。 我们也许在最终的游戏版本中不会用到这个，但是这是个在开发过程中有效的技巧。 举个例子，在测试时，即使游戏正在运行，我们也可以切换音频服务为早先提到的空服务来临时地关闭声音。
- 定位器依赖外部代码。 这是缺点。任何访问服务的代码必须假定在某处的代码已经注册过服务了。 如果没有做初始化，要么游戏会崩溃，要么服务会神秘地不工作。

- 在编译时绑定：

这里的思路是使用预处理器，在编译时间处理“定位”。就像这样：

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

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

像这样定位服务暗示了一些事情：

- 快速。 所有的工作都在编译时完成，在运行时无需完成任何东西。 编译器很可能会内联`getAudio()`调用，这是我们能达到的最快方案。
- 能保证服务是可用的。 由于定位器现在拥有服务，在编译时就进行了定位，我们可以保证游戏如果能完成编译，就不必担心服务不可用。
- 无法轻易改变服务。 这是主要的缺点。由于绑定发生在编译时，任何时候你想要改变服务，都得重新编译并重启游戏。

- 在运行时设置：

企业级软件中，如果你说“服务定位器”，他们脑中第一反应就是这个方法。 当服务被请求时，定位器在运行时做一些魔法般的事情来追踪请求的真实实现。

> **反射** 是一些编程语言在运行时与类型系统打交道的能力。 举个例子，我们可以通过名字找到类，找到它的构造器，然后创建实例。
>
> 像Lisp，Smalltalk和Python这样的动态类型语言自然有这样的特性，但新的静态语言比如C#和Java同样支持它。

通常而言，这意味着加载设置文件确认提供者，然后使用反射在运行时实例化这个类。这为我们做了一些事情：

- 我们可以更换服务而无需重新编译。 这比编译时绑定多了小小的灵活性，但是不像注册那样灵活，那里你可以真正地在运行游戏的时候改变服务。
- 非程序员也可改变服务。 这对于设计师是很好的，他们想要开关某项游戏特性，但修改源代码并不舒服。 （或者，更可能的，编程者 对设计者介入感到不舒服。）

- 同样的代码库可以同时支持多种设置。 由于从代码库中完全移出了定位处理，我们可以使用相同的代码来同时支持多种服务设置。

  这就是这个模型在企业网站上广泛应用的原因之一： 只需要修改设置，你就可以在不同的服务器上发布相同的应用。 历史上看来，这在游戏中没什么用，因为主机硬件本身是好好标准化了的， 但是很多游戏的目标是大杂烩般的移动设备，这点就很有关系了。

- 复杂。 不像前面的解决方案，这个方案是重量级的。 你得创建设置系统，也许要写代码来加载和粘贴文件，通常要做些事情来定位服务。 花时间写这些代码，就没法花时间写其他的游戏特性。

- 加载服务需要时间。 现在你会眉头紧蹙了。在运行时设置意味着你在消耗CPU循环加载服务。 缓存可以最小化消耗，但是仍暗示着在首次使用服务时，游戏需要暂停花点时间完成。 游戏开发者讨厌消耗CPU循环在不能提高游戏体验的地方。

如果服务不能被定位怎么办？

- 让使用者处理它：

  最简单的解决方案就是把责任推回去。如果定位器不能找到服务，只需返回NULL。这暗示着：

  - 让使用者决定如何掌控失败。 使用者也许在收到找不到服务的关键错误时应该暂停游戏。 其他时候可能可以安全地忽视并继续。 如果定位器不能定义全面的策略应对所有的情况，那么就将失败传回去，让每个使用者决定什么是正确的回应。

  - 使用服务的用户必须处理失败。 当然，这个的必然结果是每个使用者都必须检查服务的失败。 如果它们都以相同方式来处理，在代码库中就有很多重复的代码。 如果一百个中有一个忘了检查，游戏就会崩溃。

- 挂起游戏：

  我说过，我们不能保证服务在编译时总是可用的，但是不意味着我们不能声明可用性是游戏定位器运行的一部分。 最简单的方法就是使用断言：

```
class Locator
{
public:
  static Audio& getAudio()
  {
    Audio* service = NULL;

    // Code here to locate service...

    assert(service != NULL);
    return *service;
  }
};
```

  如果服务没有被找到，游戏停在试图使用它的后续代码之前。 这里的assert()调用没有解决无法定位服务的问题，但是它确实明确了问题是什么。 通过这里的断言，我们表明，“无法定位服务是定位器的漏洞。”

  > 如果你没见过assert()函数，单例模式一章中有解释。

  那么这为我们做了什么呢？

- 使用者不必处理缺失的服务。 简单的服务可能在成百上千的地方被使用，这节约了很多代码。 通过声明定位器永远能够提供服务，我们节约了使用者处理它的精力。
- 如果服务没有找到，游戏会挂起。 在极少的情况下，服务真的找不到，游戏就会挂起。 强迫我们解决定位服务的漏洞是好事（比如一些本该调用的初始化代码没有被调用）， 但被阻塞的所有人都得等到漏洞修复时。与大型开发团队工作时，当这种事情发生，会增加痛苦的停工时间。

- 返回空服务：

  我们在样例中实现中展示了这种修复。使用它意味着：

  - 使用者不必处理缺失的服务。 就像前面的选项一样，我们保证了总是会返回可用的服务，简化了使用服务的代码。
  - 如果服务不可用，游戏仍将继续。 这有利有弊。让我们在没有服务的情况下依然能运行游戏是很有用的。 在大团队中，当我们工作依赖的其他特性或者依赖的其他系统还没有就位时，这也是很有用的。

    缺点在于，较难查找无意缺失服务的漏洞。 假设游戏用服务去获取数据，然后基于数据做出决策。 如果我们无法注册真正的服务，代码获得了空服务，游戏也许不会像期望的那样行动。 需要在这个问题上花一些时间，才能发现我们以为可用的服务是不存在的。

    > 我们可以让空服务被调用时打印一些debug信息来缓和这点。

在这些选项中，我看到最常使用的是会找到服务的简单断言。 在游戏发布的时候，它经历了严格的测试，会在可信赖的硬件上运行。 无法找到服务的机会非常小。

在更大的团队中，我推荐使用空服务。 这不会花太多时间实现，可以减少开发中服务不可用的缺陷。 这也给你了一个简单的方式去关闭服务，无论它是有漏洞还是干扰到了现在的工作。

服务的服务范围有多大？

到目前为止，我们假设定位器给任何需要服务的地方提供服务。 当然这是这个模式的典型的使用方式，另一选项是服务范围限制到类和它的依赖类中，就像这样：

```
class Base
{
  // 定位和设置服务的代码……

protected:
  // 派生类可以使用服务
  static Audio& getAudio() { return *service_; }

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

通过这样，对服务的访问被收缩到了继承Base的类。这两种各有千秋：

- 如果全局可访问：

  - 鼓励整个代码库使用同样的服务。 大多数服务都被设计成单一的。 通过允许整个代码库接触到相同的服务，我们可以避免代码因不能获取“真正的”服务而到处实例化提供者。

    - 我们失去了何时何地使用服务的控制权。 这是让某物全局化的明显代价——任何东西都能接触它。单例 GoF模式一章讲了全局变量是多么的糟糕。

- 如果接触被限制在某个类中：
    - 我们控制了耦合。 这是主要的优点。通过显式限制服务到继承树的一个分支上，应该解耦的系统保持了解耦。
    - 可能导致重复的付出。 潜在的缺点是如果一对无关的类确实需要接触服务，每个类都要拥有服务的引用。 无论是谁定位或者注册服务，它也需要在这些类之间重复处理。

      另一个选项是改变类的继承层次，给这些类一个公共的基类，但这引起的麻烦也许多于收益。）

我的通用准则是，如果服务局限在游戏的一个领域中，那么限制它的服务范围在一个类上面。 举个例子，获取网络接口的服务可能限制于在线联网类中。 像日志这样应用更加广泛的服务应该是全局的。

参见

- 服务定位模式在很多方面是单例 GoF模式的兄弟，在应用前值得看看哪个更适合你的需求。
- Unity框架在它的`GetComponent()`方法中使用这个模式，协调它的组件模式
- 微软的XNA游戏开发框架在它的核心`Game`类中内建了这种模式。 每个实体都有一个`GameServices`对象可以用来注册和定位任何种类的服务。

© 2009-2015 Robert Nystrom