发件人：**方堃** fangkun119@icloud.com
主题：CH15 模式语言
日期：2017年5月2日 下午12:02
收件人：

# CH15 模式语言

2017年5月1日 星期一
下午3:09

模式间关系

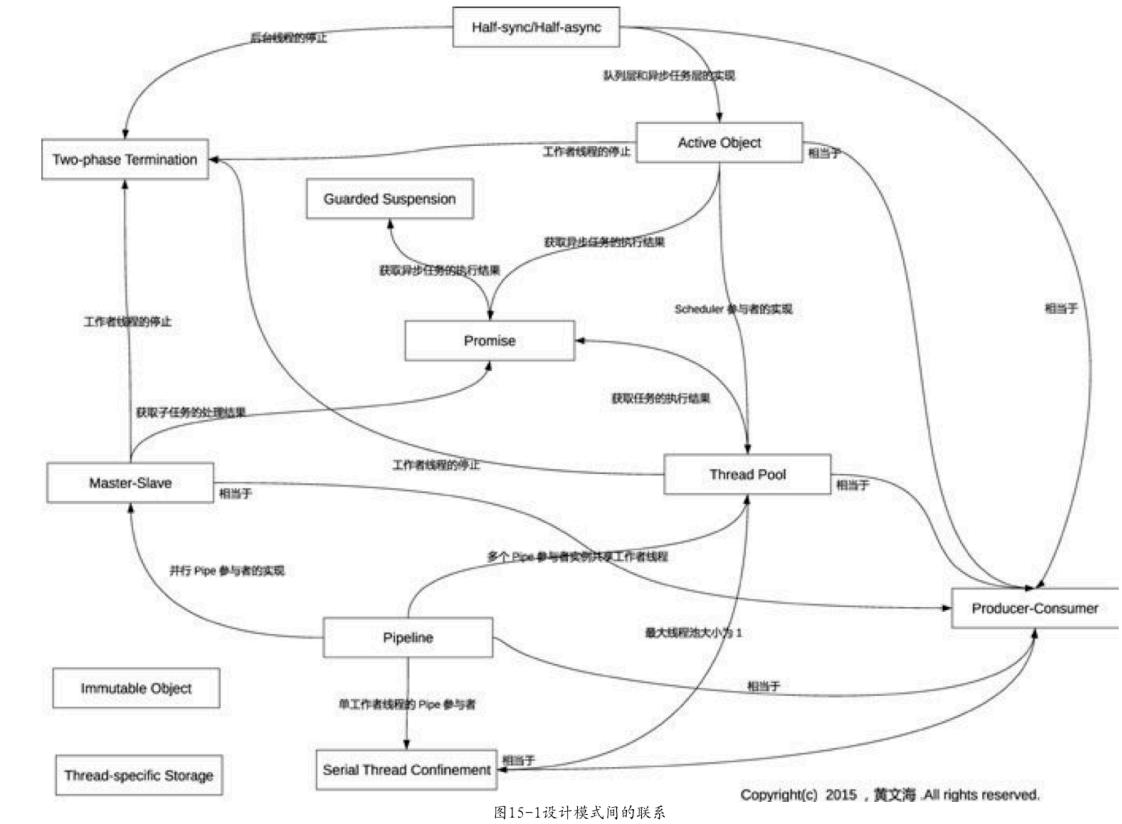


图15-1设计模式间的联系

**支持[1]（Support）。**具体的一个设计模式能够解决特定的问题，但是应用特定的设计模式本身也会引入特定的问题，而这些问题往往可以使用另外一些设计模式来解决，即在此情景下一个设计模式为另外一个设计模式解决其面临的问题提供了支持。这好比一种药物可以治疗某种疾病，但是它本身又会对人体产生一些副作用（如伤害肝脏），因此医生为病人开相应药物的时候又开了另外用于抵消该药物副作用的其他药物。例如，使用Master-Slave模式（第12章）可以提升服务的响应性，但其应用本身也会带来一些问题：Master-Slave模式中，Master参与者需要获取由各个Slave参与者实例的工作者线程执行的子任务的处理结果，而这点可以通过使用Promise模式（第6章）来实现。另外，Master参与者需要提供一个用于停止各个Slave参与者的工作者线程的接口（方法），该接口的实现可以借助Two-phase Termination模式（第5章）。

**变体（Variant）。**在特定的情况下，一个设计模式可以看作另外一个设计模式的特例（变体）。这好比正方体可以看作长和宽相等的“特殊”长方体，这里我们可以说正方体是长方体的一个变体，相应的“特定情况”是“长和宽相等”。例如，就Thread Pool模式（第9章）而言，当其最大线程池大小为1时，相应的Thread Pool模式实现就等效于一个Serial Thread Confinement模式（第11章）实现。此时，这两个模式无论是从架构上看还是从解决的问题（线程安全问题）上看都是相似的。

**组合（Combination）。**俗话说“一把钥匙开一把锁”。如果我们把一个具体的设计模式比作一把锁，而把我们要解决的问题比作要开的门，那么我们不能指望用一把锁来开启所有要开启的门。在实际解决问题的过程中，我们往往需要使用多个设计模式。在此情景下，涉及的各个设计模式之间的关系即为组合关系，即它们在特定场景下组合在一起来解决问题。例如，本书第10章（Thread Specific Storage模式）提到的实战案例中的验证码短信问题，生成随机验证码的时候我们使用了Thread Specific Storage模式以避免共享ecureRandom实例可能造成不必要等待的问题；另外，将验证码通过短信发送到用户手机的时候，我们使用了Thread Pool模式（第9章）以避免在并发用户量大的时候下发大量验证码短信导致系统创建的负责短信发送的线程过多。

**备选（Alternative）。**俗话说“条条大路通罗马”。一个（类）特定的问题往往可以采用不同的方法来解决。不同的方法采用不同的方式去解决、各自有其适用的条件和场景。例如，多线程编程经常需要解决的一个问题是线程安全的问题。本书提供了可以解决该问题的3个设计模式：Immutable Object模式（第3章）、Thread Specific Storage模式（第10章）和Serial Thread Confinement模式（第11章）。也就是说，对于线程安全问题的解决，我们有3个设计模式可选择。那么，这3个设计模式此时就形成了备选关系。

Immutable Object

## 别名

该模式也被称为Immutable（不可变）模式。

## 背景

多个线程共享一个对象的实例。

## 问题

当被共享的对象相应的现实世界实体的状态变更时，系统对此要有所反映。但是，通过直接更改该被共享的对象的状态来反映这个变更通常会导致锁的引入以保证线程安全。

### 解决方案

将相应现实世界实体建模为状态不可变的对象。当相应实现世界实体的状态变更时，系统通过创建新的对象实例来反映这种状态变更，而不是更改对象本身的状态。

### 结果

- 多个线程可以在不使用锁的情况下，以线程安全的方式去访问共享对象。
- 可能导致频繁的对象创建。

### 相关模式

Thread Specific Storage模式（第10章）和Serial Thread Confinement模式（第11章）也可以在不引入锁的情况下确保线程安全。

Guarded Suspension

## 15.3 Guarded Suspension（保护性暂挂）模式

### 别名

该模式也被称为Guarded Waits（受保护等待）模式。

### 背景

一个方法欲执行的操作（目标动作）所需的前提条件可能暂时无法满足而稍后可能得以满足。

### 问题

多线程环境中，某个对象的方法被调用时，该方法欲执行的操作所需的状态暂时没有得到满足，而稍后可能得以满足。因此，此时如果该方法返回或者抛出异常，则会迫使客户端代码对其不期望的结果进行处理。

### 解决方案

多线程环境中，当某个对象的方法（受保护方法）执行其欲执行的操作所需的状态（保护条件）未被满足时，将当前线程暂挂直到其他线程改变了该对象的状态使得保护条件得以满足时，被暂挂的线程得以唤醒。

### 结果

- 使应用程序避免了样板式（Boilerplate）代码。
- 实现了关注点分离（Separation of Concern）。
- 可能增加JVM垃圾回收的负担。
- 可能增加上下文切换（Context Switch）。

### 相关模式

Promise模式（第6章）和Producer-Consumer模式（第7章）实现过程中可能需要使用Guarded Suspension模式。

Two-phase Terminatio

## 15.4 Two-phase Termination（两阶段终止）模式

### 别名

无。

### 背景

系统需要防止用户线程（User Thread）阻止JVM正常关闭。

### 问题

守护线程（Daemon Thread）不会阻止JVM正常关闭，而用户线程会阻止JVM正常关闭。因此，正常关闭JVM时需要先将用户线程停止。但是，停止一个用户线程时，我们希望该线程能够在其处理完待处理的任务后再行停止。

### 解决方案

将线程的停止分为两个阶段：准备阶段和执行阶段。准备阶段主要实现线程停止的标志的设置，执行阶段主要实现线程停止标志的检测并在线程处理完待处理的任务后停止线程。

### 结果

- **实现了线程的优雅停止：**线程可以在其处理完待处理的任务后再行停止，而非粗暴地停止。
- **可能延迟线程的停止：**待停止的线程可能是在其处理完待处理的任务后再停止的，而这可能需要一定的等待时间。

### 相关模式

Producer-Consumer模式（第7章）和Master-Slave模式（第12章）可能需要使用Two-phase Termination模式以实现其工作者线程的停止。

Promise

## 15.5 Promise（承诺）模式

### 别名

该模式也被称为Future（期货）模式。

### 背景

一个对象需要使用另外一个对象的某个方法（以下称为目标方法）的返回值。

### 问题

目标方法需要消耗较长的处理时间才能返回表示其处理结果的值。在该方法返回之前，客户端代码会被阻塞而无法进行其他处理。

### 解决方案

使用异步编程，将目标方法的返回值改为一个凭据对象，而不是表示目标方法真正处理结果的对象（结果对象）。客户端代码通过调用凭据对象的某个方法来获取目标方法的结果对象。在此基础上，采用专门的工作者线程或者线程池去执行目标方法所进行的计算。

### 结果

- 既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性——客户端代码的编写方式与同步编程无本质差别。
- 一定程度上屏蔽了异步编程和同步编程的差异：无论目标方法是异步方法还是同步方法，它都不影响客户端代码的编写方式。

### 相关模式

目标方法可以看成Factory Method模式[2]中的工厂方法。

凭据对象用于获取结果对象的方法可能需要等待目标方法对应的计算完成才能返回，这可以使用Guarded Suspension模式（第4章）来实现。

凭据对象用于获取结果对象的方法可能需要等待相应方法对应的计算完成才能返回，这可以使用Guarded Suspension模式（第4章）来实现。

Active Object模式（第8章）可以看成是包含了Promise模式的复合模式，其异步方法的返回值就是一个凭据对象。

Producer-Consumer

## 15.6 Producer-Consumer（生产者/消费者）模式

### 别名

无。

### 背景

数据（任务）的提供方的处理能力（速率）与相应的使用方的处理能力（速率）不均衡，或者我们不希望二者的处理能力（速率）相互影响。

### 问题

数据（任务）的提供方和使用方运行在同一个线程中会导致一方处理能力（速率）的大小对另外一方产生影响，即造成等待。

### 解决方案

在数据（任务）的提供方和使用方之间引入一个作为缓冲区的通道，从而使数据（任务）的提供方和使用方可以运行在各自的线程之中。

### 结果

- 数据（任务）的提供方和使用方的处理能力相对来说互不影响。
- 关注点分离（Separation of Concern）：数据（任务）的提供方只需要将数据（任务）存入通道即可，它无须关心谁对数据（任务）进行处理；而数据（任务）的使用方只需要从通道中取出数据（任务）进行处理而无须关心是谁将其存入通道的。

### 相关模式

许多模式可看成Producer-Consumer模式的一个实例。

Active Obje

## 15.7 Active Object（主动对象）模式

### 别名

该模式也被称为Concurrent Object模式。

### 背景

客户端代码需要访问独立的线程控制（Thread of Control）对象。

### 问题

客户端代码需要使用某个类提供的服务，但是不希望等待相应的服务调用完成后才能继续其他处理，以避免响应性和吞吐率受此服务调用的影响。

### 解决方案

将服务方法的调用（Invocation）和执行（Execution）进行解耦（Decoupling）。客户端代码调用某个服务方法时，该方法并不立即执行相应的服务操作，而是生成表示相应服务操作的对象（任务）并将其存入缓存区，由专门的工作者线程取缓冲区中的任务进行执行。

### 结果

- 有利于提高并发性，从而提高系统的吞吐率。
- 使调试变得复杂。

### 相关模式

Active Object模式可看成Producer-Consumer模式（第7章）的一个实例。

Active Object模式使用了Promise模式（第6章）以实现客户端代码获取异步任务的处理结果。

Thread Poo

## 15.8 Thread Pool（线程池）模式

### 别名

无。

### 背景

多线程环境中，新的任务不断产生。

### 问题

为每个新的任务都创建一个线程去负责处理过程会导致线程不断地被创建和销毁，这会增加系统的资源消耗和上下文切换。

### 解决方案

将待处理的任务存入缓冲区，并创建一定数量的工作者线程，复用这些工作者线程使其从缓冲区中取出任务执行。

### 结果

- 抵消线程创建的开销，提高系统的响应性。
- 封装了工作者线程生命周期管理。

- 减少销毁线程的开销。
- 不恰当的使用可能导致死锁。

### 相关模式

Thread Pool模式可看成Producer-Consumer模式（第7章）的一个实例。

Thread Pool模式可以使用Two-phase Termination模式（第5章）来实现其工作者线程的停止。

Thread Spe Storage

## 15.9 Thread Specific Storage（线程特有存储）模式

### 别名

该模式也被称为Thread Local Storage模式。

### 背景

多个线程需要访问同一个非线程安全对象。或者，使用线程安全的对象，但希望能够避免其使用的锁的开销。

### 问题

多个线程访问同一个非线程安全对象（TSObject）可能产生线程安全问题，而我们又不希望因此而引入锁，以便能够避免锁的开销和相关问题。

### 解决方案

使每个线程获得一个（且仅一个）该线程所特有的TSObject实例，各个线程仅访问各自的TSObject实例，一个TSObject实例不会被多个线程共享。

### 结果

- 在不引入锁的情况下实现了对非线程安全对象访问的线程安全。
- 易于使用。
- 隐藏了系统结构，可能使系统难于理解。
- 鼓励了全局对象的使用。
- 不恰当的使用可能导致内存泄漏。

### 相关模式

Immutable Object模式（第3章）和Serial Thread Confinement模式（第11章）也能够在不引入锁的情况下确保线程安全。

Serial Thre Confinemer

## 15.10 Serial Thread Confinement（串行线程封闭）模式

### 别名

无。

### 背景

异步编程中，工作者线程需要访问非线程安全对象，而我们又不希望因此而引入锁。

### 问题

系统对某种并发任务的处理涉及非线程安全对象的访问，而我们又不希望因此而引入锁，以便能够避免锁的开销和相关问题。

### 解决方案

将并发任务通过队列串行化，再创建唯一的一个工作者线程对队列中的任务进行执行。

### 结果

- 在不引入锁的情况下实现了对非线程安全对象访问的线程安全。
- 如果客户端代码关心任务的处理结果，那么可能导致多个客户端线程等待同一个工作者线程的处理结果。

### 相关模式

Serial Thread Confinement模式可看成Producer-Consumer模式（第7章）的一个实例。

Immutable Object模式（第3章）和Thread Specific Storage模式（第10章）也能够在不引入锁的情况下确保线程安全。

如果客户端代码关心任务的处理结果，那么我们可以借用Promise模式（第6章）来实现这点。

Master-Slav

## 15.11 Master-Slave（主仆）模式

### 别名

该模式也被称为Boss-Worker（老板-伙计）模式。

### 背景

一个任务被分解为等同语义（Semantically-Identical）的若干个子任务。

### 问题

分而治之（Divide and Conquer）是解决许多问题的一个通用原则。将一个任务（原始任务）分解为若干个子任务，再让这些子任务独立执行。然后将各个子任务的处理结果组合成原始任务的处理结果。这个过程需要处理好以下几个方面。

- 客户端代码不应该知道其调用的服务是基于分而治之的计算。
- 无论是客户端代码还是子任务，它们都应该不依赖于任务分解和子任务处理结果合并的算法。

### 解决方案

在服务的客户端代码和子任务的处理之间引入一个协调性的对象（即Master）。有关分而治之的相关细节被封装在Master里面。各个子任务由专门的工作者线程负责处理。

### 结果

- **提升了计算性能：**子任务可以并行执行。
- **可交换性（Exchangeability）和可扩展性（Extensibility）：**替换某个Slave实例、增加一个Slave实例对Master参与者产生的影响很小。

### 相关模式

Master-Slave模式可看成Producer-Consumer模式（第7章）的一个实例。

Master-Slave模式中的Master参与者可能会使用Promise模式（第6章）以获取子任务的处理结果。

Pipeline

## 15.12 Pipeline（流水线）模式

### 别名

无。

### 背景

多线程编程中，规模较大的任务的处理可以分解为若干个存在依赖关系的子任务。

### 问题

规模较大的任务（原始任务）的处理可能比较耗时。如果对原始任务进行纵向分解，即分解得来的子任务中的每个任务的处理又包括若干个步骤，那么即使我们采用若干个工作者线程去负责执行子任务的执行也仍然避免不了一个子任务的处理中所出现的等待（一个处理步骤的开始要等待前一个处理步骤的完成）。

### 解决方案

对原始任务进行横向分解，即将一个任务的处理分解为若干个处理阶段（Stage），其中每个处理阶段的输出作为下一个处理阶段的输入，并且各个处理阶段都有相应的工作者线程去执行相应计算。

### 结果

- 可以对有依赖关系的任务实现并行处理。
- 为局部使用单线程模型编程提供了便利。
- 具备任务处理逻辑安排的灵活性。

### 相关模式

Pipeline模式中的处理阶段可能会使用Serial Thread Confinement模式（第11章），以实现任务处理的线程安全。

Pipeline模式可以借助Master-Slave模式（第12章）实现某个处理阶段的并行处理。

Half-sync/Half-async

## 15.13 Half-sync/Half-async（半同步/半异步）模式

### 别名

无。

### 背景

某计算同时涉及低级（或耗时较短）任务和高级（或耗时较长）任务。

### 问题

低级（或耗时较短）的任务可以直接在客户端线程中执行，但是高级（或耗时较长）任务在客户端线程中执行则会增加客户端线程的等待从而减少吞吐率并降低响应性。

### 解决方案

采用分层架构。将低级（或耗时较短）任务放在异步层由客户端线程执行，高级（或耗时较长）任务放在同步层由专门的后台工作者线程执行。异步层和同步层不直接通信，而是通过队列层进行通信。

### 结果

- 既发挥了异步编程的优势——增加系统的并发性，减少不必要的等待，又保持了同步编程的简单性。
- 各层代码可以使用独立的并发访问控制策略。

### 相关模式

Half-sync/Half-async模式可看成Producer-Consumer模式（第7章）的一个实例。

Half-sync/Half-async模式可能会使用Two-phase Termination模式（第5章）来停止后台工作者线程。

Half-sync/Half-async模式的队列层和同步层合起来可以使用Active Object模式（第8章）来实现。

Thread Pool模式（第9章）可以用来实现同步层任务的执行。

使用 Microsoft OneNote for Mac 创建。