第 9 章 在一起更好还是分开更好?

软件设计中最基本的问题之一是：给定两个功能，它们应该在同一个地方一起实现，还是应该分开实现？这个问题适用于系统中的所有层级，例如功能、方法、类和服务。例如，应该在提供面向流的文件 I/O 的类中包括缓冲，还是应该在单独的类中提供？HTTP 请求的解析应该完全在一个方法中实现，还是应该在多个方法（甚至多个类）之间划分？本章讨论做出这些决定时要考虑的因素。这些因素中的一些已经在前面的章节中进行了讨论，但是为了完整起见，这里将再次对其进行讨论。

在决定是组合还是分开时，目标是降低整个系统的复杂性并改善其模块化。可能看起来实现此目标的最佳方法是将系统划分为大量的小组件：每个单独的组件越小，组件可能就越简单。但是，细分的行为会带来额外的复杂性，而这在细分之前是不存在的：

- 一部分复杂性就来自组件的数量：组件越多，就越难以追踪所有组件，也就越难在大的组件集合中找到所需的组件。细分通常会导致更多接口，而每个新接口都会增加复杂性。
- 细分可能会导致需要附加的代码来管理组件。例如，在细分之前使用单个对象的一段代码现在可能必须管理多个对象。
- 细分会产生分离：细分后的组件将比细分前的组件距离更远。例如，在细分之前位于单个类中的方法可能在细分之后位于不同的类中，并且可能在不同的文件中。分离使开发人员更难于同时查看这些组件，甚至很难知道它们的存在。如果组件真正独立，那么分离是好的：它使开发人员可以一次专注于单个组件，而不会被其他组件分散注意力。另一方面，如果组件之间存在依赖性，则分离是不好的：开发人员最终将在组件之间来回跳转。更糟糕的是，他们可能不了解这些依赖关系，这可能导致代码缺陷。
- 细分可能导致重复：细分之前的单例代码可能需要存在于每个细分的组件中。

如果代码段紧密相关，则将它们组合在一起是最有益的。如果代码段互相无关，则最好分开。以下是判断两个代码段是否相关的一些信号：

- 它们共享信息；例如，这两段代码都可能依赖于一个特定类型文档的语法。
- 它们总是一起被使用：任何使用其中一段代码的人都可能同时使用另一段代码。这种关系形式仅在其是双向关系时才值得注意。作为反例，磁盘块的高速缓存几乎总是会涉及到哈希表，但是哈希表可以在许多不涉及磁盘块高速缓存的情况下被使用。因此，这些模块应该分开。
- 它们在概念上重叠，因为存在一个更高层级的简单类别可以涵盖这两段代码。例如，搜索子字符串和大小写转换都属于字符串操作的范畴，而流量控制和可靠的信息传递都属于网络通信的范畴。
- 不看其中的一段代码就很难理解另一段。

本章的其余部分使用更具体的规则以及示例来说明何时将代码段组合在一起以及何时将它们分开是有意义的。

9.1 如果有信息共享则组合到一起

第 5.4 节 在实现 HTTP 服务器的项目时介绍了此原则。在其第一个实现中，该项目使用了两个不同的类里的方法来读取和解析 HTTP 请求。第一个方法从网络套接字读取传入请求的文本，并将其放置在字符串对象中。第二个方法解析字符串以提取请求的各个组成部分。经过这种分解，这两个方法最终都对 HTTP 请求的格式有了相当的了解：第一个方法只是尝试读取请求，而不是解析请求，但是如果不执行大部分的解析操作，就无法确定请求的结束位置（例如，它必须解析标头行才能识别包含整个请求长度的标头）。由于此共享信息，最好在同一位置读取和解析请求；当两个类合而为一时，代码变得更短，更简单。

9.2 如果可以简化接口则组合到一起

当两个或多个模块组合成一个模块时，可以为新模块定义一个比原始接口更简单或更易于使用的接口。当原始模块各自实现问题解决方案的一部分时，通常会发生这种情况。在上一部分的 HTTP 服务器示例中，原始方法需要一个接口来从第一个方法返回 HTTP 请求字符串并将其传递给第二个方法。当这些方法结合在一起时，这些接口就不需要了。

另外，将两个或更多类的功能组合在一起时，就有可能自动执行某些功能，以至于大多数用户都无需了解它们。Java I/O 库就是展示这种机会的例子，如果将 `FileInputStream` 和 `BufferedInputStream` 类组合在一起，并且在默认情况下提供缓冲，则绝大多数用户甚至都不需要知道缓冲的存在。组合后的 `FileInputStream` 类可以提供禁用或替换默认缓冲机制的方法，但是大多数用户不需要了解它们。

9.3 通过组合来消除重复

如果发现反复重复的代码模式，请查看是否可以重新组织代码以消除重复。一种方法是将重复的代码提取为一个单独的方法，并用对该方法的调用替换重复的代码段。如果重复的代码段很长并且替换方法具有简单的签名，则此方法最有效。如果代码段只有一两行，那么用方法调用替换它可能不会有太多好处。如果代码段与其环境以复杂的方式进行交互（例如，通过访问多个局部变量），则替换方法可能需要复杂的签名（例如，许多“按引用传递”的参数），这将会降低其价值。

消除重复的另一种方法是重构代码，使相关代码段仅需要在一个地方执行。假设您正在编写一种方法，该方法需要在几个不同的执行点返回错误，并且在返回之前需要在每个执行点执行相同的清理操作（示例请参见图 9.1）。如果编程语言支持 `goto`，则可以将清除代码移到方法的最后，然后在需要返回错误的每个点处转到该片段，如图 9.2 所示。Goto 语句通常被认为是一个坏主意，如果不加选择地使用它们，可能会导致无法维护的代码，但是在诸如此类的情况下，它们可用于摆脱嵌套代码，因此也是有用的。

```
switch (common->opcode) {
    case DATA: {
        DataHeader* header = received->getStart<DataHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
```

```
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
    case GRANT: {
        GrantHeader* header = received->getStart<GrantHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
    case RESEND: {
        ResendHeader* header = received->getStart<ResendHeader>();
        if (header == NULL) {
            LOG(WARNING, "%s packet from %s too short (%u bytes)",
                    opcodeSymbol(common->opcode),
                    received->sender->toString(),
                    received->len);
            return;
        }
        ...
}
```

图 9.1：此代码处理不同类型的传入网络数据包。对于每种类型，如果数据包对于该类型而言太短，则会记录一条消息。在此版本的代码中，LOG 语句对于几种不同的数据包类型是重复的。

```
switch (common->opcode) {
    case DATA: {
        DataHeader* header = received->getStart<DataHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
    case GRANT: {
        GrantHeader* header = received->getStart<GrantHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
    case RESEND: {
        ResendHeader* header = received->getStart<ResendHeader>();
        if (header == NULL)
            goto packetTooShort;
        ...
```

```
}
...
packetTooShort:
LOG(WARNING, "%s packet from %s too short (%u bytes)",
        opcodeSymbol(common->opcode),
        received->sender->toString(),
        received->len);
return;
```

图 9.2：对图 9.1 中的代码进行了重新组织，因此只有 LOG 语句的一个副本。

9.4 分离通用代码和专用代码

如果模块包含了可用于多种不同目的的机制，则它应仅提供一种通用机制。它不应包含专门针对特定用途的机制的代码，也不应包含其他通用机制。与通用机制关联的专用代码通常应放在不同的模块中（通常是与特定用途关联的模块）。第 6 章 中的图形界面编辑器讨论阐明了这一原则：最佳设计是文本类提供通用文本操作，而特定于用户界面的操作（例如删除所选的区域）则在用户界面模块中实现。这种方法消除了早期设计中存在的信息泄露和额外的接口，而在早期设计中，专门的用户界面操作是在文本类中实现的。

危险信号：重复

> 如果相同的代码（或几乎相同的代码）一遍又一遍地出现，那是一个危险信号，说明您没有找到正确的抽象。

9.5 示例：插入光标和区域选择

接下来的章节将通过两个示例说明上述原则。在第一个示例中，最好的方法是分开相关的代码段。而在第二个示例中，最好将它们组合到一起。

第一个示例由第 6 章的图像界面编辑器项目中的插入光标和区域选择组成。编辑器会显示一个闪烁的垂直线，用来指示用户键入的文本将出现在文档中的何处。它还会显示一个高亮的字符范围，称之为选择的区域，用于复制或删除文本。插入光标始终可见，但是有时可能没有选择文本。如果存在选择的区域，则插入光标始终位于其某一端。

区域选择和插入光标在某些方面是相关的。例如，光标始终位于所选区域的一端，并且倾向于将插入光标和区域选择一起操作：单击并拖动鼠标同时修改两者，然后插入文本时会首先删除所选的文本（如果有），然后在光标位置插入新的文本。因此，使用单个对象来管理区域选择和插入光标似乎是合乎逻辑的，并且有一个项目团队就采用了这种方法。该对象在文件中存储了两个位置，以及两个布尔值，用来指示光标位于所选区域的哪一端以及是否存在区域选择。

但是，组合的对象有点尴尬。它对较高层级的代码没有任何好处，因为较高层级的代码仍然需要将区域选择和插入光标视为不同的实体，并且对它们进行单独操作（在插入文本期间，它首先在组合对象上调用一个方法来删除选定的文本；然后调用另一个方法来检索光标位置，以插入新的文本）。实际上，组合对象比分离的对象实现起来要复杂得多。它避免了将光标位置存储为单独的实体，但又不得不存储一个布尔值，以表示光标位于所选区

域的哪一端。为了检索光标位置，组合对象必须首先检查布尔值，然后再检查所选区域对应的起始或结束位置。

在这种情况下，区域选择和插入光标之间的关联度不足以将它们组合在一起。当修改代码以分开区域选择和插入光标时，用法和实现都变得更加简单。与必须从中提取所选区域和插入光标信息的组合对象相比，分开的对象提供了更简单的接口。插入光标的实现也变得更加简单，因为插入光标的位置是直接表示的，而不是通过所选区域和一个布尔值间接表示的。实际上，在修订的版本中，没有专门的类用于区域选择或插入光标。相反，引入了一个新的 `Position` 类来表示文件中的位置（行号和行内的字符数）。所选区域用两个 `Position` 表示，光标用一个 `Position` 表示。`Position` 类还在项目中找到了其他用途。这个例子也展示了第 6 章讨论过的更低层级但更通用的接口的好处。

 危险信号：通用专用混合体 

> 当通用机制还包含专门用于该机制的特定用途的代码时，就会出现此危险信号。这使该机制更加复杂，并在该机制与特定用例之间造成了信息泄露：未来对用例的修改可能也需要对底层机制进行更改。

9.6 示例：单独的日志记录类

第二个示例涉及一个学生项目中的错误日志记录。有一个类中包含几个如下所示的代码序列：

```
try {
    rpcConn = connectionPool.getConnection(dest);
} catch (IOException e) {
    NetworkErrorLogger.logRpcOpenError(req, dest, e);
    return null;
}
```

不是直接在检测到错误时记录错误日志，而是调用专门的错误日志记录类中的方法。错误日志记录类是在同一源文件的末尾定义的：

```
private static class NetworkErrorLogger {
    /**
    *  Output information relevant to an error that occurs when trying
    *  to open a connection to send an RPC.
    *
    *  @param req
    *       The RPC request that would have been sent through the
connection
    *  @param dest
    *       The destination of the RPC
    *  @param e
    *       The caught error
    */
```

```
    public static void logRpcOpenError(RpcRequest req, AddrPortTuple
dest, Exception e) {
        logger.log(Level.WARNING, "Cannot send message: " + req + ".
\n" +
            "Unable to find or open connection to " + dest + " :" +
e);
    }
...
}
```

`NetworkErrorLogger` 类包含几个方法，例如 `logRpcSendError` 和 `logRpcReceiveError`，每个方法都记录了不同类型的错误。

这种分离除了增加了复杂性，没有任何好处。日志记录方法很浅：大多数只包含一行代码，但是它们需要大量的文档。每个方法仅在单个位置调用。日志记录方法高度依赖于它们的调用方：读取调用方代码的人很可能会切换到日志记录方法，以确保记录了正确的信息。同样，阅读日志记录方法代码的人很可能会转到调用方以了解该方法的目的。

在此示例中，最好移除日志记录方法，并将日志记录语句放置在检测到错误的位置。这将使代码更易于阅读，并消除了日志记录方法所需的接口。

9.7 拆分和组合方法

何时细分的问题不仅适用于类，而且还适用于方法：是否最好将现有方法分为多个较小的方法？还是应该将两种较小的方法组合为一种较大的方法？长方法比短方法更难于理解，因此许多人认为方法长度就是拆分方法的一个很好的理由。课堂上的学生通常会获得严格的标准，例如“拆分超过 20 行的任何方法！”

但是，长度本身很少是拆分方法的一个很好的理由。通常，开发人员倾向于过多地拆分方法。拆分方法会引入额外的接口，从而增加了复杂性。它还将原始方法的各个部分分开，如果这些部分实际上是相关的，会使得代码更难阅读。您只应该在会使整个系统更加简单的情况下拆分一个方法，我将在下面讨论这种情况。

长方法并不总是坏的。例如，假设一个方法包含按顺序执行的五个 20 行的代码块。如果这些块是相对独立的，则可以一次阅读并理解该方法的一个块。将每个块移动到单独的方法中并没有太大的好处。如果这些块之间具有复杂的交互，则将它们保持在一起就显得更为重要，这样读者就可以一次看到所有代码。如果每个块使用单独的方法，则读者将不得不在这些分散开的方法之间来回切换，以了解它们如何协同工作。如果方法具有简单的签名并且易于阅读，则包含数百行代码的方法是可以接受的。这些方法很深（功能多，接口简单），很好。

设计方法时，最重要的目标是提供整洁的抽象。每个方法都应该做一件事并且完整地做这件事。该方法应该具有简单的接口，以便用户无需费神就可以正确使用它。该方法应该是深的：其接口应该比其实现简单得多。如果一个方法具有所有这些属性，那么它的长短与否就无关紧要了。

总体而言，拆分一个方法只有在会产生更清晰的抽象时才有意义。有两种方式可以做到这一点，如图 9.3 所示。最佳方法是将子任务分解为单独的方法，如图 9.3（b）所示。该细分产生一个包含该子任务的子方法和一个包含原始方法其余部分的父方法；父方法调用子方法。新的父方法的接口与原始方法的接口相同。如果存在一个与原始方法的其余部分完全可分离的子任务，则这种细分形式是有意义的，这意味着阅读子方法的人不需要了解有关父方法的任何信息，以及在阅读父方法时不需要了解子方法的实现。通常，这意味着子方法是相对通用的：可以想象除父方法外，其他方法也可以使用它。如果您进行了这种形式的拆分，然后发现自己在父方法和子方法之间来回跳转以了解它们如何一起工作，那是一个危险信号（“连体方法”），表明拆分可能不是一个好主意。

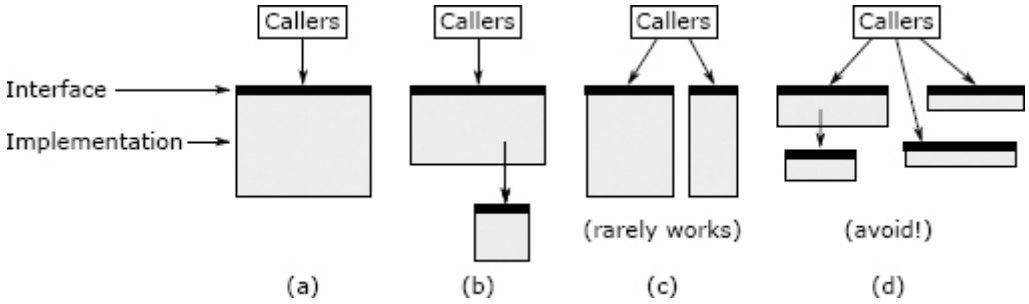

图 9.3：方法（a）可以通过提取子任务（b）或将其功能划分为两个单独的方法（c）进行拆分。但如果会导致浅方法，则不应进行方法拆分，如（d）所示。

拆分方法的第二种方法是将其拆分为两个独立的方法，每个方法都对原始方法的调用者可见，如图 9.3（c）所示。如果原始方法的接口过于复杂，这是有意义的，因为该接口试图执行多个并不密切相关的操作。在这种情况下，可以将方法的功能划分为两个或更多个较小的方法，每个方法仅具有原始方法功能的一部分。如果进行这样的拆分，则每个子方法的接口应该比原始方法的接口更简单。理想情况下，大多数调用者只需要调用两个新方法之一即可；如果调用者必须同时调用这两个新方法，则将增加复杂性，这可能表明这样的拆分不是一个好主意。新方法将更加专注于它们自己的工作。如果新方法比原始方法更具通用性，那么这是一个好兆头（例如，您可以想象在其他情况下单独使用它们）。

图 9.3（c）所示形式的拆分并不是很有意义，因为它们导致调用者不得不处理多个方法而不是一个方法。当您以这种方式拆分时，您可能会遇到变成多个浅方法的风险，如图 9.3（d）所示。如果调用者必须调用每个单独的方法，并在它们之间来回传递状态，则拆分不是一个好主意。如果您正在考虑像图 9.3（c）所示的拆分，则应基于它是否简化了调用者的使用情况来进行判断。

在某些情况下，通过将方法组合在一起可以简化系统。例如，组合方法可以用一个更深的方法代替两个浅的方法。它可以消除重复的代码；它可以消除原始方法或中间数据结构之间的依赖关系；它可以产生更好的封装，从而使以前存在于多个位置的知识现在被隔离在一个位置；它也可以产生更简单的接口，如 9.2 节所述。

🚩危险信号：连体方法🚩

> 应该可以独立地理解每一个方法。如果您只能在理解一个方法的实现的前提下才能理解另一个方法的实现，那就是一个危险信号。该危险信号也可以在其他情况下发生：如果两段代码在物理上是分开的，但是只有通过查看另一段代码才能理解它们，这就是危险信号。

9.8 不同的观点：整洁的代码

在《整洁代码之道》一书中 [1]，Robert Martin 认为函数应该仅根据长度进行拆分。他说，函数应该非常短，甚至 10 行都太长了：

> 函数的第一规则是要短小，第二条规则是还要更短小... if 语句、else 语句、while 等语句块中的代码应该只有一行，该行大抵是一个函数调用语句... 这也意味着函数不应该大到足以容纳嵌套结构。所以，函数的缩进层级不应多于一层或两层。当然，这样的函数易于阅读和理解。

我同意短的函数一般来说比长的函数更容易理解。然而，一旦函数的代码行数少到几十行，进一步的减少行数不太可能对可读性产生太大的影响。更重要的是，函数的分解是否降低了系统的整体复杂性？换句话说，阅读几个短函数并理解它们是如何协同工作的比阅读一个较大的函数更容易吗？更多的函数意味着更多的接口需要文档化和学习。如果函数太小了，它们就失去了独立性，导致必须一起阅读和理解的连体函数。当这种情况发生时，最好还是保留较大的函数，以便所有相关的代码都在一个地方。深度比长度更重要：首先使函数变深，然后尝试使它们足够短，以便易于阅读。不要为了长度而牺牲深度。

9.9 结论

拆分或组合模块的决定应基于复杂性。请选择一种可以提供最好的信息隐藏、最少的依赖关系和最深的接口的结构。

[1] 整洁代码之道, Robert C. Martin, Pearson Education, Inc., Boston, MA 2009