第 18 章 代码应该是易理解的

模糊性是第 2.3 节中描述的造成复杂性的两个主要原因之一。当系统的重要信息对于新开发人员而言并不明显时，就会产生模糊性。解决模糊性的方法是以容易理解的方式编写代码。本章讨论了一些使代码更易理解或更难理解的因素。

如果代码是易理解的，则意味着人们可以快速阅读其代码，无需多想，他们对代码行为或含义的猜测也将是正确的。如果代码是易理解的，那么读者就不需要花费太多时间或精力来收集他们使用代码所需的所有信息。如果代码是难理解的，那么读者必须花费大量时间和精力来弄懂它。这不仅会降低他们的效率，而且还增加了误解和引入缺陷的可能性。易理解的代码比难理解的代码需要的注释更少。

这里说的"易理解"是对读者而言的：注意到别人代码里难以理解的地方比发现自己的代码有问题要容易得多。因此，确定代码是否易理解的最佳方法是通过代码审查。如果阅读您代码的人说它不是易理解的，那么它就不是易理解的，无论它对您来说是多么的清晰。通过尝试理解什么使代码变得难理解，您将学会如何在未来写出更好的代码。

18.1 使代码更易理解的事情

在前面的章节中已经讨论了使代码易理解的两种最重要的技术。首先是选取好名称（第 14 章）。精确而有意义的名称可以阐明代码的行为，并减少对文档的需求。如果名称含糊不清，那么读者将不得不通读代码，以推论被命名实体的含义，而这既费时又容易出错。第二种技术是一致性（第 17 章）。如果总是以相似的方式完成相似的事情，那么读者可以识别出他们以前所见过的模式，并立即得出（安全）结论，而无需详细分析代码。

以下是其它一些使代码更易理解的通用技术：

明智地使用空格。代码的格式化方式会影响其被理解的容易程度。考虑以下参数文档，其中空格已被压缩：

```
/**
 *  ...
 *  @param numThreads The number of threads that this manager should
 *  spin up in order to manage ongoing connections. The MessageManager
 *  spins up at least one thread for every open connection, so this
 *  should be at least equal to the number of connections you expect
 *  to be open at once. This should be a multiple of that number if
 *  you expect to send a lot of messages in a short amount of time.
 *  @param handler Used as a callback in order to handle incoming
 *  messages on this MessageManager's open connections. See
 *  {@code MessageHandler} and {@code handleMessage} for details.
 */
```

很难看到一个参数的文档在哪里结束而下一个参数的文档在哪里开始。甚至不知道有多少个参数或它们的名称是什么。如果添加了一些空格，结构会突然变得清晰，文档也更容易阅读：

```
/**
 *  @param numThreads
 *           The number of threads that this manager should spin up in
 *           order to manage ongoing connections. The MessageManager
spins
 *           up at least one thread for every open connection, so this
 *           should be at least equal to the number of connections you
 *           expect to be open at once. This should be a multiple of
that
 *           number if you expect to send a lot of messages in a short
 *           amount of time.
 *  @param handler
 *           Used as a callback in order to handle incoming messages
on
 *           this MessageManager's open connections. See
 *           {@code MessageHandler} and {@code handleMessage} for
details.
 */
```

空白行也可用于分隔方法中的主要代码块，例如以下示例：

```
void* Buffer::allocAux(size_t numBytes) {
    // Round up the length to a multiple of 8 bytes, to ensure
alignment.
    uint32_t numBytes32 =  (downCast<uint32_t>(numBytes) + 7) & ~0x7;
    assert(numBytes32 != 0);

    // If there is enough memory at firstAvailable, use that. Work
down
    // from the top, because this memory is guaranteed to be aligned
    // (memory at the bottom may have been used for variable-size
chunks).
    if (availableLength >= numBytes32) {
        availableLength -= numBytes32;
        return firstAvailable + availableLength;
    }

    // Next, see if there is extra space at the end of the last chunk.
    if (extraAppendBytes >= numBytes32) {
        extraAppendBytes -= numBytes32;
        return lastChunk->data + lastChunk->length + extraAppendBytes;
    }

    // Must create a new space allocation; allocate space within it.
    uint32_t allocatedLength;
    firstAvailable = getNewAllocation(numBytes32, &allocatedLength);
    availableLength = allocatedLength numBytes32;
```

```
    return firstAvailable + availableLength;
}
```

如果每个空白行之后的第一行是描述下一个代码块的注释，则此方法特别有效：空白行使注释更可见。

语句中的空格有助于阐明语句的结构。比较以下两个语句，其中之一具有空格，而另外一个没有空格：

```
for(int pass=1;pass>=0&&!empty;pass--) {

for (int pass = 1; pass >= 0 && !empty; pass--) {
```

注释。有时无法避免难以理解的代码。发生这种情况时，重要的是使用注释来提供缺少的信息以进行弥补。要做好这一点，您必须把自己放在读者的位置上，弄清楚什么可能会使他们感到困惑，以及哪些信息可以消除这种困惑。下一节将介绍几个示例。

18.2 使代码难理解的事情

有很多事情可以使代码变得难理解。本节提供了一些示例。其中一些，例如事件驱动编程，在某些情况下很有用，所以您可能最终会使用它们。发生这种情况时，额外的文档有助于最大程度地减少读者的困惑。

事件驱动编程。在事件驱动编程中，应用程序对外部事件做出响应，例如网络数据包的到来或鼠标按钮被按下。一个模块负责报告传入的事件，而应用程序的其他部分通过注册感兴趣的事件来要求事件模块在事件发生时调用给定的函数或方法。

事件驱动编程使得控制流程很难被跟踪。事件处理函数从未被直接调用，它们是由事件模块间接调用的，通常使用函数指针或接口。即使您在事件模块中找到了调用点，也仍然无法确定哪个具体的函数会被调用：这将取决于在运行时注册了哪些处理程序。正因为如此，很难对事件驱动的代码进行推理，也很难说服自己它是在有效工作的。

为了弥补这种模糊性，使用每个处理函数的接口注释来表明它何时被调用，如以下示例所示：

```
/**
 * This method is invoked in the dispatch thread by a transport if a
 * transport-level error prevents an RPC from completing.
 */
void Transport::RpcNotifier::failed() {
    ...
}
```

危险信号：难理解的代码

> 如果无法通过快速阅读来理解代码的含义和行为，则它是一个危险信号。通常，这意味着阅读代码的人并不能立即搞清楚某些重要的信息。

通用容器。许多语言提供了用于将两个或多个项目组合到一个对象中的通用类，例如 Java 中的 `Pair` 或 C++ 中的 `std::pair`。这些类很诱人，因为它们使得通过单个变量传递多个对象变得容易。最常见的用途之一是从一个方法返回多个值，如以下 Java 示例所示：

```
return new Pair<Integer, Boolean>(currentTerm, false);
```

不幸的是，通用容器会导致代码不清晰，因为分组后的元素具有含义模糊的通用名称。在上面的示例中，调用者必须使用 `result.getKey()` 和 `result.getValue()` 引用两个返回的值，这并没有提供关于这些值的实际含义的任何线索。

因此，最好不要使用通用容器。如果需要容器，请定义新的专门用于特定用途的类或结构。然后，您可以为元素使用有意义的名称，并且可以在声明中提供额外的文档，而对于通用容器而言这些都是不可能的。

此示例说明了一条通用规则：**软件应被设计成易于阅读的而不是易于编写的。**通用容器对于编写代码的人来说是很方便的，但是它们会给所有后续的读者带来困惑。对于编写代码的人来说，花一些额外的时间来定义特定的容器结构是更好的选择，这样写出来的代码更容易理解。

在声明和赋值中使用了不同的类型。考虑以下 Java 示例：

```
private List<Message> incomingMessageList;
...
incomingMessageList = new ArrayList<Message>();
```

该变量被声明为 `List`，但实际的值类型为 `ArrayList`。这段代码是合法的，因为 `List` 是 `ArrayList` 的超类，但是它会误导只看到声明没看到实际赋值的读者。实际的类型可能会影响变量的使用方式（`ArrayList` 与 `List` 的其他子类相比，具有不同的性能和线程安全属性），因此最好让声明与赋值的类型互相匹配。

超出读者期望的代码。考虑以下代码，这是 Java 应用程序的主程序：

```
public static void main(String[] args) {
    ...
    new RaftClient(myAddress, serverAddresses);
}
```

大多数应用程序在其主程序返回时退出，因此读者可能会认为这里也是一样的。然而，事实并非如此。`RaftClient` 的构造函数创建了额外的线程，即使应用程序的主线程结束了，该线程仍在继续运行。应该在 `RaftClient` 构造函数的接口注释中记录此行为，但是该行为不够明显，因此值得在 `main` 函数的末尾添加简短注释，该注释应描述该应用程序将继续在其他线程中执行。如果代码符合读者期望的约定，那么它是易理解的。如果不是，那么将行为记录下来就很重要，这样读者才不会感到困惑。

18.3 结论

关于易理解性的另一种思考方式是从信息的角度出发。如果代码难理解，则通常意味着代码还存在读者不了解的重要信息：在 `RaftClient` 示例中，读者可能不知道 `RaftClient` 构造函数创建了新线程；在 `Pair` 示例中，读者可能不知道 `result.getKey()` 返回当前项的编号。

为了使代码容易理解，您必须确保读者总是拥有理解代码所需的信息。您可以通过三种方式来做到这一点。最好的方法是使用抽象和消除特殊情况等设计技术，以减少需要了解的信息量。其次，您可以利用读者在其他上下文中已经了解到的信息（例如，通过遵循约定并符合期望），这样读者不必为您的代码去了解新的信息。第三，您可以使用诸如好名称和战略式注释之类的技术在代码中向他们提供重要的信息。