第 13 章 注释应该描述代码中难以理解的内容

编写注释的原因是，使用编程语言编写的语句无法表达编写代码时开发人员想到的所有重要信息。注释记录了这些信息，以便后来的开发人员可以轻松地理解和修改代码。注释的指导原则是，**注释应该描述代码中难以理解的内容。**

从代码来看，有许多事情并不容易理解。有时，是底层细节不容易理解。例如，当用一对索引描述一个范围时，由索引给出的元素是在范围之内还是之外并不明显。有时不清楚为什么需要一些代码，或者为什么要以特定方式实现代码。有时，开发人员遵循一些规则，例如 “总是在 b 之前调用 a”。您可能可以通过查看所有代码来猜测出规则，但这很痛苦且容易出错。注释可以使规则清晰明了。

写注释的最重要原因之一是抽象，其中包括许多从代码中看不到的信息。抽象的思想是提供一种思考问题的简单方法，但是代码是如此详细，以至于很难仅通过阅读代码就看到抽象。注释可以提供一个更简单、更高层级的视图（比如：调用此方法后，网络流量将被限制为每秒 `maxBandwidth` 字节）。即使可以通过阅读代码推断出此信息，我们也不想强迫模块用户这样做：阅读代码很耗时，并且会迫使他们考虑很多使用该模块不需要的信息。**开发人员无需阅读除其外部可见声明以外的任何代码就应该能够理解模块提供的抽象。** 唯一的方法是用注释来补充声明。

本章讨论需要在注释中描述哪些信息以及如何编写良好的注释。就像您将看到的那样，好的注释通常以与代码不同的详细程度来解释事物，在某些情况下，注释会更详细，而在某些情况下，会不那么详细（更抽象）。

13.1 选择约定

编写注释的第一步是确定注释的约定，例如您要注释的内容和注释的格式。如果您正在使用已经有文档编译工具的语言进行编程，例如 Java 的 Javadoc， C++ 的 Doxygen 或 Go! 的 godoc，请遵循这些工具的约定。这些约定都不是完美的，但是这些工具可提供足够的好处来弥补这一缺点。如果在没有现有约定可遵循的环境中进行编程，请尝试采用其他类似的语言或项目中的约定，这将使其他开发人员更容易理解和遵守您的约定。

约定有两个目的。首先，它们确保一致性，这使得注释更易于阅读和理解。其次，它们有助于确保您实际编写注释。如果您不清楚要注释的内容以及写注释的方式，那么最终很容易根本不写注释。

大多数注释属于以下类别之一：

接口：在模块声明（例如类、数据结构、函数或方法）之前的注释块。该注释描述模块的接口。对于一个类，注释描述了该类提供的整体抽象。对于方法或函数，注释描述其整体行为、其参数和返回值（如果有）、其产生的任何副作用或异常、以及调用者在调用该方法之前必须满足的任何其他要求。

数据结构成员：数据结构中字段声明旁边的注释，例如类的实例变量或静态变量。

实现注释：方法或函数代码内部的注释，它描述代码在内部的工作方式。

跨模块注释：描述跨模块边界的依赖项的注释。

最重要的注释是前两个类别中的注释。每个类都应有一个接口注释，每个类变量应有一个注释，每个方法都应有一个接口注释。有时，变量或方法的声明是如此明显，以至于在注释中添加任何内容都没有实际帮助（`getter` 和 `setter` 有时都属于此类），但这很少见。注释所有这些内容要比花精力担心是否需要注释容易得多。具体实现的注释通常是不必要的（请参阅下面的第 13.6 节）。跨模块注释是最罕见的，而且编写起来很成问题，但是当需要它们时，它们就很重要。第 13.7 节将更详细地讨论它们。

13.2 不要重复代码

不幸的是，许多注释并不是特别有用。最常见的原因是注释重复了代码：可以轻松地从注释旁边的代码中推断出注释中的所有信息。这是最近研究论文中出现的代码示例：

```
ptr_copy = get_copy(obj)              # Get pointer copy
if is_unlocked(ptr_copy):             # Is obj free?
    return obj                        # return current obj
if is_copy(ptr_copy):                 # Already a copy?
    return obj                        # return obj
thread_id = get_thread_id(ptr_copy)
if thread_id == ctx.thread_id:        # Locked by current ctx
    return ptr_copy                   # Return copy
```

这些注释中几乎没有任何有用的信息，除了 `Locked by`，该注释暗示了某些线程相关的信息可能在代码中并不明显。请注意，这些注释的详细程度与代码大致相同：每行代码有一个注释，用于描述该行。这样的注释基本没用。

以下是注释重复了代码的更多示例：

```
// Add a horizontal scroll bar
hScrollBar = new JScrollBar(JScrollBar.HORIZONTAL);
add(hScrollBar, BorderLayout.SOUTH);

// Add a vertical scroll bar
vScrollBar = new JScrollBar(JScrollBar.VERTICAL);
add(vScrollBar, BorderLayout.EAST);

// Initialize the caret-position related values
caretX     = 0;
caretY     = 0;
caretMemX  = null;
```

这些注释均未提供任何价值。对于前两个注释，代码已经很清楚了，它实际上不需要注释。第三个注释可能有用，但是当前注释没有提供足够的细节来提供帮助。

编写注释后，请问自己以下问题：从未看过代码的人能否仅通过查看注释旁边的代码来写出这样的注释？如果答案是肯定的（如上述示例所示），则注释不会使代码更易于理解。像这样的注释是为什么有些人认为注释毫无价值的原因。

另一个常见的错误是在注释中使用与被注释实体相同名称的词：

```
/*
 * Obtain a normalized resource name from REQ.
 */
private static String[] getNormalizedResourceNames(HTTPRequest req)
...
/*
 * Downcast PARAMETER to TYPE.
 */
private static Object downCastParameter(String parameter, String type)
...
/*
 * The horizontal padding of each line in the text.
 */
private static final int textHorizontalPadding = 4;
```

 危险信号：注释重复了代码 

> 如果注释中的信息可以很明显的从旁边的代码中看出，则注释是没有帮助的。这样的一个例子是，当注释使用与所描述事物名称相同的单词时。

这些注释只是从方法或变量名中提取单词，或者从参数名称和类型中添加几个单词，然后将它们组成一个句子。例如，第二个注释中唯一不在代码中的是单词 `to`！再一次，这些注释可以仅通过查看声明来编写，无需对变量的方法有任何了解，所以它们没有价值。

同时，注释中缺少一些重要信息：例如，什么是规范化的资源名称（normalized resource name）？`getNormalizedResourceNames` 返回的数组的元素是什么？“downcast” 是什么意思？填充（padding）的单位是什么，是仅在每行的一边填充还是两边都填充？在注释中描述这些内容将很有帮助。

编写良好注释的第一步是 **在注释中使用与被描述实体名称不同的词**。为注释选择单词，以提供有关实体含义的更多信息，而不仅仅是重复其名称。例如，以下是针对 textHorizontalPadding 的更好注释：

```
/*
 * The amount of blank space to leave on the left and
 * right sides of each line of text, in pixels.
 */
private static final int textHorizontalPadding = 4;
```

该注释提供了从声明本身看不出来的额外信息，例如单位（像素）以及填充适用于每行两边的事实。考虑到读者可能不熟悉术语“填充”，注释没有直接使用这个术语，而是解释了什么是填充。

13.3 更低层级的注释可提高精确度

现在您知道了不应该做的事情，让我们讨论应该在注释中添加哪些信息。**注释通过提供不同详细程度的信息来增强代码。** 一些注释提供了比代码更低层级、更详细的信息。这些注释通过阐明代码的确切含义来增加精确度。其他注释提供了比代码更高层级、更抽象的信息。这些注释反映了直觉，例如代码背后的考量，或者更简单、更抽象的代码思考方式。与代码处于同一层级的注释可能会重复该代码。本节将详细地讨论更低层级的方式，而下一节将讨论更高层级的方式。

在注释变量声明（例如类实例变量、方法参数和返回值）时，精确度最有用。变量声明中的名称和类型通常不是很精确。注释可以填写缺少的详细信息，例如：

- 此变量的单位是什么？
- 边界条件是包含还是排除？
- 如果允许使用空值，那么它意味着什么？
- 如果变量引用了最终必须释放或关闭的资源，那么谁负责释放或关闭该资源？
- 是否存在某些对于变量始终不变的属性（不变量），例如“此列表始终包含至少一个条目”？

这部分信息可以通过检查使用该变量的所有代码推断出来。但是，这很耗时且容易出错。声明的注释应清晰和完整，使读者没必要通过检查使用该变量的所有代码来了解这些信息。另外，当我说声明的注释应描述代码中难以理解的内容时，这里的“代码”是指注释旁边的代码（即声明），而不是“应用程序中的所有代码”。

变量注释最常见的问题是注释太模糊。这是两个不够精确的注释示例：

```
// Current offset in resp Buffer
uint32_t offset;

// Contains all line-widths inside the document and
// number of appearances.
private TreeMap<Integer, Integer> lineWidths;
```

在第一个示例中，`Current` 的含义不清晰。在第二个示例中，不清楚 `TreeMap` 中的键是不是线宽、值是不是出现次数。另外，宽度是以像素或字符为单位吗？以下修订后的注释提供了更多详细信息：

```
//  Position in this buffer of the first object that hasn't
//  been returned to the client.
uint32_t offset;

//  Holds statistics about line lengths of the form <length, count>
//  where length is the number of characters in a line (including
//  the newline), and count is the number of lines with
//  exactly that many characters. If there are no lines with
//  a particular length, then there is no entry for that length.
private TreeMap<Integer, Integer> numLinesWithLength;
```

第二个声明使用一个较长的名称 `numLinesWithLength` 来传达更多信息。它还将“宽度（Width）”更改为“长度（Length）”，因为该术语更可能使人们认为单位是字符而不是像素。请注意，第二条注释不仅记录了每个条目的详细信息，还记录了缺失条目的含义。

在为变量添加注释时，请考虑使用名词而不是动词。换句话说，关注变量代表什么，而不是如何被操纵。考虑以下注释：

```
/* FOLLOWER VARIABLE: indicator variable that allows the Receiver and
the
 * PeriodicTasks thread to communicate about whether a heartbeat has
been
 * received within the follower's election timeout window.
 * Toggled to TRUE when a valid heartbeat is received.
 * Toggled to FALSE when the election timeout window is reset.
 */
private boolean receivedValidHeartbeat;
```

该文档描述了如何通过类中的几段代码来修改变量。如果注释描述变量代表什么而不是重复代码逻辑，则注释将更短且更有用：

```
/* True means that a heartbeat has been received since the last time
 * the election timer was reset. Used for communication between the
 * Receiver and PeriodicTasks threads.
 */
private boolean receivedValidHeartbeat;
```

基于该文档，很容易推断出，当接收到心跳信号时，变量必须设置为真；而当重置选举计时器时，则必须将变量设置为假。

13.4 更高层级的注释可增强直觉

注释可以增加代码可读性的第二种方式是提供直觉。这些注释是在比代码更高的层级上编写的。它们忽略了细节，并帮助读者理解了代码的整体意图和结构。这种方式通常用于方法内部的注释以及接口注释。例如，考虑以下代码：

```
// If there is a LOADING readRpc using the same session
// as PKHash pointed to by assignPos, and the last PKHash
// in that readRPC is smaller than current assigning
// PKHash, then we put assigning PKHash into that readRPC.
int readActiveRpcId = RPC_ID_NOT_ASSIGNED;
for (int i = 0; i < NUM_READ_RPC; i++) {
    if (session == readRpc[i].session
            && readRpc[i].status == LOADING
            && readRpc[i].maxPos < assignPos
            && readRpc[i].numHashes < MAX_PKHASHES_PERRPC) {
        readActiveRpcId = i;
        break;
    }
}
```

该注释太底层也太详细。一方面，它部分重复了代码：比如 `if there is a LOADING readRPC` 只是重复了 `readRpc[i].status == LOADING`。另一方面，注释没能解释代码的总体目的，也不能解释其在包含它的代码中的作用。如此一来，这个注释不能帮助读者理解代码。

这是一个更好的注释：

```
// Try to append the current key hash onto an existing
// RPC to the desired server that hasn't been sent yet.
```

此注释不包含任何详细信息。相反，它在更高层级上描述了代码的整体功能。有了这些高层级的信息，读者就可以解释代码中发生的几乎所有事情：循环一定是在遍历所有已经存在的远程过程调用（RPC）；会话测试可能用于查看特定的 RPC 是否发往正确的服务器；`LOADING` 测试表明 RPC 可以具有多个状态，在某些状态下添加更多的哈希值是不安全的；`MAX_PKHASHES_PERRPC` 测试表明在单个 RPC 中可以发送多少个哈希值是有限制的。注释中唯一没有解释的是 `maxPos` 测试。此外，新注释为读者评估代码提供了基础：它可以完成将哈希密钥添加到一个现有 RPC 所需的一切吗？而原始的注释并未描述代码的整体意图，因此，读者很难确定代码是否行为正确。

更高层级的注释比更低层级的注释更难编写，因为您必须以不同的方式考虑代码。问问自己：这段代码要做什么？您能以哪种最简单的方式来解释代码中的所有内容？这段代码最重要的是什么？

工程师往往非常注重细节。我们喜欢细节，善于管理其中的许多细节，这对于成为一名优秀的工程师至关重要。但是，优秀的软件设计师也可以从细节退后一步，从更高的层级考虑系统。这意味着要确定系统的哪些方面最重要，并且能够忽略底层细节，仅根据系统的最基本特征来考虑系统。这是抽象的本质（找到一种思考复杂实体的简单方法），这也是您在编写更高层级注释时必须要做的。一个好的更高层级注释表达了一个或几个简单的想法，这些想法提供了一个概念框架，例如“追加到现有的 RPC”。使用该框架，可以很容易地看到特定的代码语句与总体目标之间的关系。

这是另一个代码示例，具有更高层级的注释：

```
if (numProcessedPKHashes < readRpc[i].numHashes) {
    // Some of the key hashes couldn't be looked up in
    // this request (either because they aren't stored
    // on the server, the server crashed, or there
    // wasn't enough space in the response message).
    // Mark the unprocessed hashes so they will get
    // reassigned to new RPCs.
    for (size_t p = removePos; p < insertPos; p++) {
        if (activeRpcId[p] == i) {
            if (numProcessedPKHashes > 0) {
                numProcessedPKHashes--;
            } else {
                if (p < assignPos)
                    assignPos = p;
                activeRpcId[p] = RPC_ID_NOT_ASSIGNED;
            }
        }
    }
}
```

此注释做了两件事。第二句话提供了代码功能的抽象描述。而第一句话是不同的：它以高层级的方式解释了为什么要执行这些代码。类似于“我们是如何到达这里的”的注释对于帮助人们理解代码非常有用。例如，在为方法添加注释时，描述最有可能在什么情况下调用该方法（特别是仅在非正常场景下才需要调用该方法的时候）会非常有帮助。

13.5 接口文档

注释最重要的作用之一就是定义抽象。回想一下第 4 章，抽象是实体的简化视图，它保留了基本信息，但省略了可以安全忽略的细节。代码不适合描述抽象，它的层级太低，它包含了不应该在抽象中看到的实现细节。描述抽象的唯一方法是使用注释。**如果您想要呈现良好抽象的代码，则必须用注释记录这些抽象。**

文档化抽象的第一步是将接口注释与实现注释分开。接口注释提供了使用类或方法时需要知道的信息，它们定义了抽象。实现注释则描述了类或方法如何在内部工作以实现抽象。区分这两种注释很重要，这样就不会对接口的用户暴露实现细节。此外，这两种形式最好有所不同。**如果接口注释也必须描述实现，则该类或方法是浅的。** 这意味着编写注释的行为可以提供有关设计质量的线索，第 15 章将回到这个想法。

类的接口注释提供了该类提供的抽象的高层级描述，例如：

```
/**
 * This class implements a simple server-side interface to the HTTP
 * protocol: by using this class, an application can receive HTTP
 * requests, process them, and return responses. Each instance of
 * this class corresponds to a particular socket used to receive
 * requests. The current implementation is single-threaded and
 * processes one request at a time.
 */
public class Http {...}
```

该注释描述了类的整体功能，没有任何实现细节，甚至没有特定方法的细节。它还描述了该类的每个实例代表什么。最后，注释描述了该类的限制（它不支持从多个线程的并发访问），这对于考虑是否使用它的开发人员可能很重要。

方法的接口注释既包括用于抽象的高层级信息，又包括用于精确度的低层级细节：

- 注释通常以一两个句子开头，描述调用者能感知到的方法的行为。这是更高层级的抽象。
- 注释必须描述每个参数和返回值（如果有）。这些注释必须非常精确，并且必须描述对参数值的任何约束以及参数之间的依赖关系。
- 如果该方法有任何副作用，则必须在接口注释中记录这些副作用。副作用是该方法对系统的未来行为的影响，但又不是结果的一部分。例如，如果该方法将一个值添加到内部数据结构中，可以通过将来的方法调用来检索该值，则这是副作用。写入文件系统也是一种副作用。
- 方法的接口注释必须描述该方法可能产生的任何异常。
- 如果有任何在调用方法之前必须满足的前提条件，则必须对其进行描述（也许必须先调用其他方法；对于二分查找方法，被查找的列表必须是已排序的）。尽量减少前提条件是一个好主意，但是任何留下来的都必须记录在案。

这是一个从 `Buffer` 对象复制数据的方法的接口注释：

```
/**
 * Copy a range of bytes from a buffer to an external location.
 *
 * \param offset
 *        Index within the buffer of the first byte to copy.
 * \param length
 *        Number of bytes to copy.
 * \param dest
 *        Where to copy the bytes: must have room for at least
 *        length bytes.
 *
 * \return
 *        The return value is the actual number of bytes copied,
 *        which may be less than length if the requested range of
 *        bytes extends past the end of the buffer. 0 is returned
```

```
 *        if there is no overlap between the requested range and
 *        the actual buffer.
 */

uint32_t
Buffer::copy(uint32_t offset, uint32_t length, void* dest)
...
```

此注释的语法（例如 `\return`）遵循 Doxygen 的约定，该程序从 C / C++ 代码中提取注释并将其编译为 Web 页面。注释的目的是提供开发人员调用该方法所需的所有信息，包括特殊情况的处理方式（请注意此方法是如何遵循第 10 章的建议并通过定义来规避与范围指定相关的任何错误的）。开发人员不必为了调用它而阅读方法的主体，并且接口注释也没有提供关于如何实现该方法的任何信息，比如它是如何扫描其内部数据结构以查找所需的数据。

接下来是一个更全面的示例，让我们考虑一个称为 `IndexLookup` 的类，该类是分布式存储系统的一部分。存储系统拥有一个表集合，每个表包含许多对象。另外，每个表可以具有一个或多个索引；每个索引都基于对象的特定字段提供对表中对象的高效访问。例如，一个索引可以用于根据对象的名称字段查找对象，而另一个索引可以用于根据对象的年龄字段查找对象。使用这些索引，应用程序可以快速提取具有特定名称或者具有给定范围内的年龄的所有对象。

`IndexLookup` 类为执行索引查找提供了一个方便的接口。这是一个如何在应用程序中使用它的示例：

```
query = new IndexLookup(table, index, key1, key2);
while (true) {
    object = query.getNext();
    if (object == NULL) {
        break;
    }
    ... process object ...
}
```

应用程序首先构造一个 `IndexLookup` 类型的对象，并提供用于选择表、索引和索引范围的参数（例如，如果索引基于年龄字段，则 `key1` 和 `key2` 可以指定为 21 和 65 选择年龄介于这些值之间的所有对象）。然后，应用程序重复调用 `getNext` 方法。每次调用都返回一个位于所需范围内的对象。一旦返回所有匹配的对象，`getNext` 将返回 `NULL`。因为存储系统是分布式的，所以此类的实现有些复杂。表中的对象可以分布在多个服务器上，每个索引也可以分布在一组不同的服务器上。`IndexLookup` 类中的代码必须首先与所有相关的索引服务器通信，以收集指定范围内对象的信息，然后必须与实际存储对象的服务器通信，以检索它们的值。

现在，让我们考虑该类的接口注释中需要包含哪些信息。对于下面给出的每条信息，问问自己，开发人员是否需要知道该信息才能使用该类（我对这些问题的回答在本章的结尾）：

1. `IndexLookup` 类发送给索引服务器和对象服务器的消息格式。
2. 用于确定特定对象是否在所需范围内的比较功能（比较是使用整数、浮点数还是字符串来完成的？）。
3. 用于在服务器上存储索引的数据结构。
4. `IndexLookup` 是否同时向多个服务器发出多个请求。
5. 处理服务器崩溃的机制。

这是 `IndexLookup` 类的接口注释的原始版本；摘录还包括了类定义里的几行内容，将在注释中被引用到：

```
/*
 * This class implements the client side framework for index range
 * lookups. It manages a single LookupIndexKeys RPC and multiple
 * IndexedRead RPCs. Client side just includes "IndexLookup.h" in
 * its header to use IndexLookup class. Several parameters can be set
 * in the config below:
 * - The number of concurrent indexedRead RPCs
 * - The max number of PKHashes a indexedRead RPC can hold at a time
 * - The size of the active PKHashes
 *
 * To use IndexLookup, the client creates an object of this class by
 * providing all necessary information. After construction of
 * IndexLookup, client can call getNext() function to move to next
 * available object. If getNext() returns NULL, it means we reached
 * the last object. Client can use getKey, getKeyLength, getValue,
 * and getValueLength to get object data of current object.
 */
class IndexLookup {
    ...
private:
    /// Max number of concurrent indexedRead RPCs
    static const uint8_t NUM_READ_RPC = 10;
    /// Max number of PKHashes that can be sent in one
    /// indexedRead RPC
    static const uint32_t MAX_PKHASHES_PERRPC = 256;
    /// Max number of PKHashes that activeHashes can
    /// hold at once.
    static const size_t MAX_NUM_PK = (1 << LG_BUFFER_SIZE);
 }
```

在进一步阅读之前，看看你是否能找出这个注释的问题。这些是我发现的问题：

- 第一段的大部分与实现有关，而不是接口。举一个例子，用户不需要知道用于与服务器通信的特定远程过程调用的名称。在第一段的后半部分中提到的配置参数都是私有变量，它们仅与类的维护者相关，而与类的用户无关。所有这些实现信息都应从注释中省略。

- 该注释还包括一些显而易见的事情。例如，不需要告诉用户包括 `IndexLookup.h`：任何编写 C++ 代码的人都可以猜测这是必要的。另外，“通过提供所有必要的信息（by providing all necessary information）”实际上什么也没说，因此也可以省略。

一个更简短的注释对这个类就足够了（并且更可取）：

```
/*
 * This class is used by client applications to make range queries
 * using indexes. Each instance represents a single range query.
 *
 * To start a range query, a client creates an instance of this
 * class. The client can then call getNext() to retrieve the objects
 * in the desired range. For each object returned by getNext(), the
 * caller can invoke getKey(), getKeyLength(), getValue(), and
 * getValueLength() to get information about that object.
 */
```

此注释的最后一段不是严格必需的，因为它主要是对几个方法的注释的重复。但是，在类文档中提供示例来说明其方法如何协同工作可能会有所帮助，特别是对于使用模式不明显的深类尤其如此。注意，新的注释未提及 `getNext` 的 `NULL` 返回值。此注释无意记录每个方法的每个细节；它只是提供高层级的信息，以帮助读者了解这些方法如何协同工作以及何时可以调用每个方法。读者可以参考各个方法的接口注释来了解更多细节。此注释也没有提到服务器崩溃；这是因为服务器崩溃对于该类的用户是不可见的（系统会自动从中恢复）。

 危险信号：实现文档污染了接口 

> 当接口文档（例如方法的文档）记录了使用过程中不需要知道的详细实现信息时，就会出现此危险信号。

现在考虑以下代码，该代码显示了 `IndexLookup` 类中 `isReady` 方法的文档的第一版：

```
/**
 * Check if the next object is RESULT_READY. This function is
 * implemented in a DCFT module, each execution of isReady() tries
 * to make small progress, and getNext() invokes isReady() in a
 * while loop, until isReady() returns true.
 *
 * isReady() is implemented in a rule-based approach. We check
 * different rules by following a particular order, and perform
 * certain actions if some rule is satisfied.
 *
 * \return
 *         True means the next Object is available. Otherwise, return
 *         false.
 */
bool IndexLookup::isReady() { ... }
```

同样的问题，本文档中的大多数内容，例如对 `DCFT` 的引用以及整个第二段，都与实现有关，因此不应该放在这里，这是接口注释中最常见的错误之一。某些实现文档很有用，但应放在方法内部，应将其与接口文档明确分开。此外，文档的第一句话是含糊的（`RESULT_READY` 是什么意思？），并且缺少一些重要信息。最后，无需在此处描述 `getNext` 的实现。这是该注释的更好版本：

```
/*
 * Indicates whether an indexed read has made enough progress for
 * getNext to return immediately without blocking. In addition, this
 * method does most of the real work for indexed reads, so it must
 * be invoked (either directly, or indirectly by calling getNext) in
 * order for the indexed read to make progress.
 *
 * \return
 *         True means that the next invocation of getNext will not
block
 *         (at least one object is available to return, or the end of
the
 *         lookup has been reached); false means getNext may block.
 */
```

此注释版本提供了更精确的关于“就绪（ready）”的信息，还提供了一个重要信息：如果要继续进行索引检索，则最终必须调用此方法。

13.6 实现注释：做什么以及为什么这么做，而不是如何做

实现注释是出现在方法内部的注释，用来帮助读者了解它们在内部的工作方式。大多数方法是如此简短，简单，以至于它们不需要任何实现注释：有了代码和接口注释，就很容易弄清楚方法的工作原理。

实现注释的主要目的是帮助读者理解代码在做什么（而不是代码如何工作）。一旦读者知道了代码要做什么，通常就很容易理解代码的工作原理。对于简短的方法，代码只做一件事，既然已经在其接口注释中进行了描述，就不需要实现注释了。较长的方法具有多个代码块，这些代码块作为方法的整体任务的一部分执行不同的操作。在每个主要块之前添加注释，以提供对该块的作用的高层级（更抽象）描述。这是一个例子：

```
// Phase 1: Scan active RPCs to see if any have completed.
```

这样的注释可以帮助读者在代码中找到他们关心的部分。对于循环语句，在循环前加一个注释来描述每次迭代中发生的事情是有帮助的：

```
// Each iteration of the following loop extracts one request from
// the request message, increments the corresponding object, and
// appends a response to the response message.
```

请注意此注释如何更抽象和直观地描述循环。它没有详细介绍如何从请求消息中提取请求或对象如何递增。仅对于更长或更复杂的循环才需要循环注释，在这种情况下，循环的作用可能并不明显。许多循环足够短且简单，以至于其行为已经很明显。

除了描述代码在做什么之外，实现注释还有助于解释为什么这么做。如果代码中有些复杂的地方很难直接从代码中看出来，则应将它们记录下来。例如，如果一个缺陷修复需要添加目的不是很明显的代码，请添加注释以说明为什么需要该代码。对于缺陷修复，如果已经有缺陷报告很好地描述了这个问题，该注释可以引用缺陷跟踪数据库中的问题编号，而不是重复其所有详细信息（比如“修复 RAM-436，与 Linux 2.4.x 中的设备驱动程序崩溃有关。）。开发人员可以在缺陷数据库中查找更多详细信息（这是一个避免注释重复的示例，这将在第 16 章中进行讨论）。

对于更长的方法，为一些最重要的局部变量写注释会有帮助。但是，如果大多数局部变量都有比较好的名称，也不需要文档。如果变量的所有用法在几行代码之内都是可见的，则通常无需注释即可轻松理解变量的用途。在这种情况下，可以让读者阅读代码来弄清楚变量的含义。但是，如果在大量代码中使用了该变量，则应考虑添加注释以描述该变量。在对变量进行文档化时，应关注变量表示的内容，而不是代码中如何对其进行操作。

13.7 跨模块设计决策

在理想环境中，每个重要的设计决策都将封装在一个类中。不幸的是，真实系统中的设计决策不可避免地最终会影响到多个类。例如，网络协议的设计将影响发送方和接收方，并且它们可以在不同的地方实现。跨模块决策通常是复杂而微妙的，并且容易导致代码缺陷，因此，为它们提供良好的文档至关重要。

跨模块文档的最大挑战是找到一个放置它的位置，以便开发人员能自然地发现它。有时，会有一个明显的中心位置用于放置此类文档。例如，RAMCloud 存储系统定义了一个状态值，每个请求均返回该状态值以指示成功或失败。为新的错误状况添加状态需要修改许多不同的文件（一个文件将状态值映射到异常，另一个文件为每个状态提供人类可读的消息，等等）。幸运的是，在添加新的状态值时，有一个显而易见的地方是开发人员必须去的，那就是状态枚举的声明。我们利用了这一点，在该枚举中添加了注释，以标识所有其他必须同步修改的地方。

```
typedef enum Status {
    STATUS_OK = 0,
    STATUS_UNKNOWN_TABLET                = 1,
    STATUS_WRONG_VERSION                 = 2,
    ...
    STATUS_INDEX_DOESNT_EXIST            = 29,
    STATUS_INVALID_PARAMETER             = 30,
    STATUS_MAX_VALUE                     = 30,
    // Note: if you add a new status value you must make the following
    // additional updates:
    // (1)  Modify STATUS_MAX_VALUE to have a value equal to the
    //      largest defined status value, and make sure its definition
    //      is the last one in the list. STATUS_MAX_VALUE is used
```

```
    //      primarily for testing.
    // (2)  Add new entries in the tables "messages" and "symbols" in
    //      Status.cc.
    // (3)  Add a new exception class to ClientException.h
    // (4)  Add a new "case" to ClientException::throwException to map
    //      from the status value to a status-specific ClientException
    //      subclass.
    // (5)  In the Java bindings, add a static class for the exception
    //      to ClientException.java
    // (6)  Add a case for the status of the exception to throw the
    //      exception in ClientException.java
    // (7)  Add the exception to the Status enum in Status.java,
making
    //      sure the status is in the correct position corresponding
to
    //      its status code.
}
```

新的状态值将添加到现有列表的末尾，因此注释也放在了最有可能被看到的末尾。

不幸的是，在许多情况下，并没有一个明显的中心位置用来放置跨模块文档。RAMCloud存储系统中的一个例子是处理僵尸服务器的代码，僵尸服务器是系统认为已经崩溃但实际上仍在运行的服务器。使僵尸服务器无效需要几个不同模块中的代码，这些代码都相互依赖。没有一段代码是明显的放置文档的中心位置。一种可能性是在每个依赖文档的位置复制部分的文档。然而，这是不合适的，并且随着系统的演进，很难使这样的文档保持最新。或者，文档可以位于需要它的位置之一，但是在这种情况下，开发人员不太可能看到文档或者知道在哪里查找它。

我最近一直在尝试一种方法，该方法将跨模块问题记录在一个名为 `designNotes` 的中央文件中。该文件分为几个清晰标识的部分，每个部分针对一个主题。例如，以下是该文件的摘录：

```
...
Zombies
-------
A zombie is a server that is considered dead by the rest of the
cluster; any data stored on the server has been recovered and will
be managed by other servers. However, if a zombie is not actually
dead (e.g., it was just disconnected from the other servers for a
while) two forms of inconsistency can arise:
* A zombie server must not serve read requests once replacement
   servers have taken over; otherwise it may return stale data that
   does not reflect writes accepted by the replacement servers.
* The zombie server must not accept write requests once replacement
   servers have begun replaying its log during recovery; if it does,
   these writes may be lost (the new values may not be stored on the
   replacement servers and thus will not be returned by reads).
```

```
RAMCloud uses two techniques to neutralize zombies. First,
...
```

然后，在与这些问题之一相关的任何代码段中，都有一条简短的注释引用了 `designNotes` 文件：

```
// See "Zombies" in designNotes.
```

使用这种方法，文档只有一个副本，因此开发人员在需要时可以相对容易地找到它。但是，这样做的缺点是，文档离依赖它的任何代码段都不近，因此随着系统的演进，可能难以保持最新。

13.8 结论

注释的目的是确保系统的结构和行为对读者来说是易理解的，因此他们可以快速找到所需的信息，并有信心对系统进行修改，确信这些修改能够正常工作。某些信息可以通过代码本身直观地展现给读者，但是还有大量信息无法从代码中轻易推断出来。注释将补充这些信息。

当遵循注释应描述代码中难以理解的内容的规则时，是否"难以理解"是从第一次阅读您代码的人（而不是您自己）的角度出发的。在编写注释时，请尝试使自己进入读者的心态，并问自己他或她需要知道哪些关键事项。如果您的代码正在接受审核，并且审核者告诉您某些内容难以理解，请不要与他们争论。如果读者认为它难以理解，那么它就是难以理解的。与其争论，不如尝试了解他们发现的令人困惑的地方，并看看是否可以通过更好的注释或更好的代码来澄清它们。

13.9 回答第 13.5 节中的问题

开发人员是否需要了解以下每条信息才能使用 `IndexLookup` 类？

1. `IndexLookup` 类发送给索引服务器和对象服务器的消息格式。否：这是应该隐藏在类中的实现细节。
2. 用于确定特定对象是否在所需范围内的比较功能（比较是使用整数、浮点数还是字符串来完成的？）。是：该类的用户需要了解此信息。
3. 用于在服务器上存储索引的数据结构。否：此信息应封装在服务器上；甚至 `IndexLookup` 的实现都不需要知道这一点。
4. `IndexLookup` 是否同时向多个服务器发出多个请求。有可能：如果 `IndexLookup` 使用特殊技术来提高性能，则文档应提供相关的一些高层级信息，因为用户可能会在意性能。
5. 处理服务器崩溃的机制。否：RAMCloud 可从服务器崩溃中自动恢复，因此崩溃对于应用程序级软件不可见；因此，在 `IndexLookup` 的接口文档中无需提及崩溃。如果崩溃反映到应用程序中，则接口文档将需要描述它们会如何表现出来（而不是崩溃恢复如何工作的详细信息）。