【软件构造】第八章第三节 代码调优的设计模式和**I/O**

第八章第三节 代码调优的设计模式和**I/O**

本节学习如何通过对代码的修改，消除性能瓶颈，提高系统性能？——代码调优、面向性 能的设计模式

Outline

- Java调优
  - 代码调优的概念
  - 单例模式（Singleton Pattern）
  - 享元模式（Flyweight Pattern）
  - 原型模式（Prototype Pattern）
  - 对象池模式（Object Pool Pattern）
- 常见的Java I/O方法

Notes

代码调优

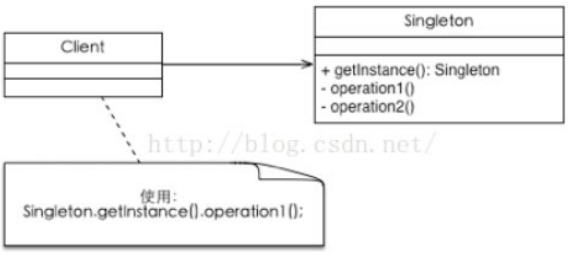
【代码调优的概念】

- 代码调优：代码调优不是为了修复bug，而是对正确的代码进行修改以提高其性能，其常常是小规模的变化
  - 调优不会减少代码行数
  - 不要猜原因，而应有明确的优化目标
  - 不要边写程序边调优
  - 不是性能优化的第一选择
  - 代码行数与性能之间无必然的联系
  - 代码调优建立在对程序性能的精确度量基础之上（profiling）
  - 当程序做过某些调整之后，要重新profiling并重新了解需要优化的性能瓶颈，微小的变化能导致优化方向大不相同
- 性能从不是追求的第一目标，正确性比性能更重要

【单例模式（**Singleton Pattern**）】

- 定义：某些类在应用运行期间只需要一个实例。
- 现状：某些类在运行时只要需要一个实例就new，导致很多情况下创建多个object。
- 更好的选择：强制client只能创建一个object实例，避免因为new操作所带来的时空性能（尤其是GC）的损失，也便于复用。
- 优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。2、避免对资源的多重占用（比如写文件操作）。
- 缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。
- 实现：
  - 对重构的代码进行封装，只提供一个访问点

    - 提供static的方法允许指定调用
- 模式图：

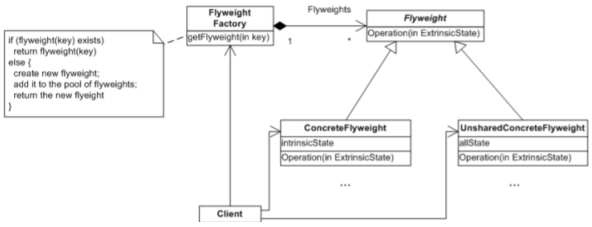


- 注意：
    - 1. 只创建一个实例，并且只提供一个全局的访问点；避免创建多个实例的可能。
    - 2. 资源共享情况下，获取实例的方法必须适应多线程并发访问。
    - 3. 提高访问性能。
    - 4. 懒加载（Lazy Load），在需要的时候才被构造。

```
public class Singleton {
  private static final Singleton instance = null;
  private Singleton() {…}
  public static Singleton getInstance() {
      if (instance == null)
            instance = new Singleton();
      return instance;
   }
   // other operations and data
}
```

【享元模型（Flyweight Pattern）】

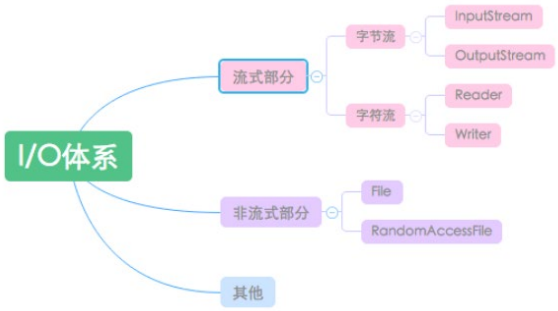
- 使用场景： 1、系统有大量相似对象。 2、需要缓冲池的场景。
- 特点：该模式允许在应用 中不同部分共享使用objects，降低大量objects带来的时空代价
- 对象的内部状态：不管在什么场合使用该object，内部特征都不变。
- 对象的外部状态：不是固定的，需要在不同场合context分别指派/计算其值。
- 实现：
    - flyweight声明一个接口来接受外部状态并采取行动
    - ConcreteFlyweight保存可共享状态，UnsharedConcreteFlyweight不可共享
    - FlyweightFactory负责flyweight的创建、管理、提供给客户端
    - 客户端通过索引获取flyweight对象，调用方法计算外在状态
- 优点：大大减少对象的创建，降低系统的内存，使效率提高。
- 缺点：提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱
- 模式图：



【原型模式（Prototype Pattern）】

【对象池模式（Object Pool Pattern）】

常见Java I/O



具体请移步 **Java I/O 总结**

- Java中I/O操作主要是指使用Java进行输入，输出操作. Java所有的I/O机制都是基于数据流进行输入输出，这些数据流表示了字符或者字节数据的流动序列。
- 虽然java IO类库庞大，但总体来说其框架还是很清楚的。从是读媒介还是写媒介的维度看，Java IO可以分为：
  - 输入流：InputStream和Reader
  - 输出流：OutputStream和Writer
- 而从其处理流的类型的维度上看，Java IO又可以分为：
  - 字节流：InputStream和OutputStream
  - 字符流：Reader和Writer
- 下面这幅图就清晰的描述了JavaIO的分类：

-	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

① InputStream/Reader

InputStream/Reader 是输入流，这里的输入输出是相对于内存来说的。这两个类都是基类（抽象类），不能实例化对象，可以靠它的派生类来实例化对象，实现文件的各种操作。InputStream 是字节流，Reader 是字符流。

② OutputStream/Writer

OutputStream/Writer 是输出流，这里是指输出到磁盘等存储介质上，这两个类也都是基类，可以用它们的派生类来实现各种操作。OutputStream 是字节流，Writer 是字符流。

【如何选择**I/O**流】

- 确定是输入还是输出
  - 输入:输入流 InputStream Reader
  - 输出:输出流 OutputStream Writer
- 明确操作的数据对象是否是纯文本
  - 是:字符流 Reader，Writer
  - 否:字节流 InputStream，OutputStream
- 明确具体的设备。
  - 文件:

  读：FileInputStream,, FileReader,

  写：FileOutputStream，FileWriter
  - 数组:

  byte[ ]：ByteArrayInputStream, ByteArrayOutputStream

  char[ ]：CharArrayReader, CharArrayWriter
  - String:

  StringBufferInputStream(已过时，因为其只能用于String的每个字符都是8位的字符串), StringReader, StringWriter
  - Socket流

  键盘：用System.in（是一个InputStream对象）读取，用System.out（是一个OutoutStream对象）打印
- 是否需要转换流
  - 是，就使用转换流，从Stream转化为Reader、Writer:
  - InputStreamReader，OutputStreamWriter
- 是否需要缓冲提高效率
  - 是就加上Buffered：BufferedInputStream, BufferedOuputStream, BufferedReader, BufferedWriter
- 是否需要格式化输出