文章编号:1005-1228(2016)06-0017-03

多核多线程下 java 设计模式性能提升

赵思焰
(爱迪德技术(北京)有限公司 北京 100125)

摘 要:现代大型系统中多核多线程下的应用越来越多,java 语言发展至今对于并行机制有了很大改善,架构中的设计模式从性能的角度也在发生着变化,文章着重介绍几个典型设计模式进行探讨,以便在系统重构或开发中得到性能改进。以及 JDK 在设计模式上改进的期待。
关键词:Java 性能;设计模式;多线程;多核
中图分类号:TP311.1 **文献标识码**:A

Java Design Pattern for Performance Optimization in Multithreading and Multi-core Programming

ZHAO Si-yan
(Irdeto Technology(Beijing) Co Ltd. Beijing 100125 ,China)

Abstract:More and more modern large-scale web applications are applied in multithreading and multi-core processor model system.Up to now Java language mechanism has already improved a lot for parallel and concurrency. From architecture view, the design patterns for java should be updated as well. From this article, it will introduce several typical design patters which improve performance and are applied for module development and refactor. At the same time that JDK is improved for design patter would be expected in future.
Key words: Java performance; design patter; multithreading; multi-core processor

在多线程下怎样使 Java 模式更好的支持多核 CPU 既是语言的发展也是设计模式发展的动力。此篇着重分析几个典型模式下对多核多线程设计模式的应用,从而提升系统性能。

1 单例模式的性能提升

单例模式应用很多,线程池,数据库连接池等,为了性能的提升既要保持单一对象的数据同步,又要实现多线程无锁访问。

1.1 方法一 内部类延迟加载

```
public class InnerClassForSingleton {
    private InnerClassForSingleton(){
        System.out.println("InnerClassForSingleton is created");
    }
    private static class SingletonHolder{
        private static InnerClassForSingleton instance = new
        InnerClassForSingleton();
    }
    public static InnerClassForSingleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

代码中因为内部类 SingletonHolder 和内部变量 instance 同为 static 达到同步无锁化的目的,而不必实例化 InnerClassForSingleton

1.2 方法二 单例对象的原子化:利用 volatile 实现多线程对单例实例的共享

```
public class DCLPatternForSinglton {

    private volatile static DCLPatternForSinglton dclForSinglton;
    private DCLPatternForSinglton() {
        System.out.println("DCLPatternForSinglton is created");
    }
    public static DCLPatternForSinglton getInstance() {
```

收稿日期:2016-09-05
作者简介:赵思焰(1973-),男,北京人,首席工程师,主要研究方向:大数据及 Java 在大型系统中性能提升研究。

```
        try {
            if (dclForSinglton != null) {
                // do nothing
            } else {
                synchronized (DCLPatternForSinglton.class) {
                    if (dclForSinglton == null) {
                        dclForSinglton = new DCLPat-
ternForSinglton();
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return dclForSinglton;
    }
}
```

Volatile 总是得到修改最后的值，相当于原子操作，即当对象锁定被实例化后即通知所有线程实例修改，达到无锁化访问。此方法性能略低于方法一，性能等级一样。

这里也希望将来 java 的原子化对象更加丰富，提升模式性能

2 代理模式的性能提升

代理模式在开源框架中应用非常频繁，例如 spring 的 aop，hibernate 的 O/R mapping 等都是代理模式的核心功能应用，此模式一方面提倡即插即用的灵活性，另一方面屏蔽网络或 IO 的各种开销达到性能优化。JDK 的 Proxy 原生代理类和 CGLIB 都能实现代理模式，尽管代理模式可以易于实现延迟加载，但最终实例化对象最终决定性能的优略，这里推荐使用 CGLIB，但若实现接口则仍然需用 Proxy。

```
public static IDBQuery createCglibProxy() {
    Enhancer enhancer = new Enhancer();
// createCglibProxy is a innerclass
    enhancer.setCallback(p.new CglibDbQueryInterceptor());
    enhancer.setInterfaces(new Class[] { IDBQuery.class });
    IDBQuery cglibProxy = (IDBQuery) enhancer.create();
    return cglibProxy;
}
```

也希望 JDK 以后的版本提高 Proxy 的执行效率。

3 享元模式

此模式的核心思想是对于多个相同的对象，只需共享一份拷贝。从而减少对象的创建和减小 GC 的压力。例如 spring 中 FactoryBean<Object>，对可 singleton 的对象利用 getObject()取得共享对象，同时更好的管理对象内部方法和属性。模式 code 举例如下：

```
public interface IReportManager {
    public String createReport();
}
public class FinancialReportManager implements IReportMan-
ager {
    protected String tenantId = null;

    public FinancialReportManager(String tenantId) {
        this.tenantId = tenantId;
    }
    @Override
    public String createReport( ) {
        return "This is a financial report";
    }
}
public class EmployeeReportManager implements IReport-
Manager {
    protected String tenantId = null;
    public EmployeeReportManager(String tenantId) {
        this.tenantId = tenantId;
    }
    @Override
    public String createReport( ) {
        return "This is a employee report";
    }
}
public class ReportManagerFactory {
    Map<String, IReportManager> financialReportManager =
new HashMap<String, IReportManager>( );
    Map<String, IReportManager> employeeReportManager
= new HashMap<String, IReportManager>( );
```

ReportManagerFactory 工厂将内部 Report 类分离共享，采用参数 tenantId 作为单条记录共享。实现了共享的灵活性

4 装饰者模式

其核心思想利用动态添加对象或 Handler，对系统解耦或动态叠加加载，多线程下有效的细粒度调用从而不过多产生无效对象或垃圾资源，例如 spring 中定义 <aop:scoped-proxy/>，则会触发 AopNamespaceHandler 类调用 ScopedProxyBeanDefinitionDecorator 类的 decorate 方法解析 scope 对象，AopNamespaceHandler 则起到了用时加载和动态增加的目的。示例代码如下：

```
public interface IPacketCreator {
    public String handleContent( );
```

```
    }

    public class PacketBodyCreator implements IPacketCreator {
        @Override
        public String handleContent() {
            return "Content of Packet";
        }
    }

    public abstract class PacketDecorator implements IPacketCre-
ator {
        IPacketCreator componet;

        public PacketDecorator(IPacketCreator c) {
            componet = c;
        }
    }

    public class PacketHTMLHeaderCreator extends PacketDeco-
rator {

        public PacketHTMLHeaderCreator(IPacketCreator c) {
            super(c);
        }

        @Override
        public String handleContent() {
            StringBuffer sb = new StringBuffer();
            sb.append("<html>");
            sb.append("<body>");
            sb.append(componet.handleContent());
            sb.append("</body>");
            sb.append("</html>\n");
            return sb.toString();
        }
    }

    public class PacketHTTPHeaderCreator extends PacketDeco-
rator {
        public PacketHTTPHeaderCreator(IPacketCreator c) {
            super(c);
        }

        @Override
        public String handleContent() {
            StringBuffer sb = new StringBuffer();
            sb.append("Cache-Control:no-cache\n");
            sb.append          ("Date:Mon,31Dec201204:25:
57GMT\n");
            sb.append(componet.handleContent());
            return sb.toString();
        }
    }
```

PacketHTTPHeaderCreator,PacketHTMLHeaderCreator,PacketBodyCreator 都实现了 handleContent 方法,且 PacketHTMLHeaderCreator 和 PacketHTTPHeaderCreator 都有着 super 的引用关系,对粒度的伸缩性管理。

5　结束语

Java 性能提升模式不仅仅只有如上提及,更多模式优化可参看 https://github.com/skyscreen/java/tree/master/resume/java/optimzation,总之充分利用服务端 CPU 资源并行并发的处理能力,努力有效减少 GC 回收开销,以及粒度引用,池,缓存的有效使用,充分体现设计模式在应用中优化特性才会得到一个性能提升的系统。

参考文献:

[1] Spring Framework[EB/OL].http://projects.spring.io/spring-framework/.
[2] Hibernate[EB/OL]. http://hibernate.org/orm/.
[3] 葛一鸣等.Java 程序性能优化[M].清华大学出版社,2012.