WIKIPEDIA

双重检查锁定模式

维基百科，自由的百科全书

双重检查锁定模式（也被称为"双重检查加锁优化"，"锁暗示"（Lock hint）[1]) 是一种软件设计模式用来减少并发系统中竞争和同步的开销。双重检查锁定模式首先验证锁定条件(第一次检查)，只有通过锁定条件验证才真正的进行加锁逻辑并再次验证条件(第二次检查)。

该模式在某些语言在某些硬件平台的实现可能是不安全的。有的时候，这一模式被看做是反模式。

它通常用于减少加锁开销，尤其是为多线程环境中的单例模式实现“惰性初始化”。惰性初始化的意思是直到第一次访问时才初始化它的值。

目录

Java中的使用

考虑下面的Java代码[3] (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)

```
// Single threaded version
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }

  // other functions and members...
}
```

这段在使用多线程的情况下无法正常工作。在多个线程同时调用getHelper()时，必须要获取锁，否则，这些线程可能同时去创建对象，或者某个线程会得到一个未完全初始化的对象。

锁可以通过代价很高的同步来获得，就像下面的例子一样。

```
// Correct but possibly expensive multithreaded version
class Foo {
  private Helper helper = null;
  public synchronized Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }

  // other functions and members...
}
```

只有getHelper()的第一次调用需要同步创建对象，创建之后getHelper()只是简单的返回成员变量，而这里是无需同步的。 由于同步一个方法会降低100倍或更高的性能[2]， 每次调用获取和释放锁的开销似乎是可以避免的：一旦初始化完成，获取和释放锁就显得很不必要。许多程序员一下面这种方式进行优化：

1. 检查变量是否被初始化(不去获得锁)，如果已被初始化立即返回这个变量。
2. 获取锁
3. 第二次检查变量是否已经被初始化：如果其他线程曾获取过锁，那么变量已被初始化，返回初始化的变量。
4. 否则，初始化并返回变量。

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized(this) {
        if (helper == null) {
          helper = new Helper();
        }
      }
    }
    return helper;
  }

  // other functions and members...
}
```

直觉上，这个算法看起来像是该问题的有效解决方案。然而，这一技术还有许多需要避免的细微问题。例如，考虑下面的事件序列：

1. 线程*A*发现变量没有被初始化, 然后它获取锁并开始变量的初始化。
2. 由于某些编程语言的语义，编译器生成的代码允许在线程*A*执行完变量的初始化之前，更新变量并将其指向部分初始化的对象。

3. **线程*B*发现共享变量已经被初始化，并返回变量。由于线程*B*确信变量已被初始化，它没有获取锁。如果在*A*完成初始化之前共享变量对*B*可见（这是由于*A*没有完成初始化或者因为一些初始化的值还没有覆盖*B*使用的内存(缓存一致性)），程序很可能会崩溃。**

在J2SE 1.4或更早的版本中使用双重检查锁有潜在的危险，有时会正常工作：区分正确实现和有小问题的实现是很困难的。取决于编译器，线程的调度和其他并发系统活动，不正确的实现双重检查锁导致的异常结果可能会间歇性出现。重现异常是十分困难的。

在J2SE 5.0中，这一问题被修正了。volatile关键字保证多个线程可以正确处理单件实例。[4](http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)描述了这一新的语言特性:

```
// Works with acquire/release semantics for volatile
// Broken under Java 1.4 and earlier semantics for volatile
class Foo {
  private volatile Helper helper = null;
  public Helper getHelper() {
    Helper result = helper;
    if (result == null) {
      synchronized(this) {
        result = helper;
        if (result == null) {
          helper = result = new Helper();
        }
      }
    }
    return result;
  }

  // other functions and members...
}
```

注意局部变量result的使用看起来是不必要的。对于某些版本的Java虚拟机，这会使代码提速25%，而对其他的版本则无关痛痒。[3]

如果helper对象是静态的(每个类只有一个), 可以使用双重检查锁的替代模式惰性初始化模式[4]。查看[5] 上的列表16.6。

```
// Correct lazy initialization in Java
@ThreadSafe
class Foo {
  private static class HelperHolder {
    public static Helper helper = new Helper();
  }

  public static Helper getHelper() {
    return HelperHolder.helper;
  }
}
```

这是因为内部类直到他们被引用时才会加载。

Java 5中的final语义可以不使用volatile关键字实现安全的创建对象：[6]

```
public class FinalWrapper<T> {
  public final T value;
  public FinalWrapper(T value) {
    this.value = value;
  }
}

public class Foo {
  private FinalWrapper<Helper> helperWrapper = null;

  public Helper getHelper() {
    FinalWrapper<Helper> wrapper = helperWrapper;

    if (wrapper == null) {
      synchronized(this) {
        if (helperWrapper == null) {
          helperWrapper = new FinalWrapper<Helper>(new Helper());
        }
        wrapper = helperWrapper;
      }
    }
    return wrapper.value;
  }
}
```

为了正确性，局部变量wrapper是必须的。这一实现的性能不一定比使用volatile的性能更高。

Microsoft Visual C++ 中的使用

如果指针是由C++关键字**volatile**定义的，那么双重检查锁可以在Visual C++ 2005 或更高版本中实现。Visual C++ 2005 保证volatile变量是一种内存屏障，阻止编译器和CPU重新安排读入和写出语义。[7] 在先前版本的Visual C++则没有此类保证。在其他方面将指针定义为volatile可能会影响程序的性能。例如，如果指针定义对代码的其他地方可见，强制编译器将指针视为屏障，就会降低程序的性能，这是完全不必要的。

参见

- 测试和测试并设置idiom作为底层加锁机制。
- 按需初始化持有者作为Java中线程安全的替代者。

参考资料

1. Schmidt, D et al. Pattern-Oriented Software Architecture Vol 2, 2000 pp353-363
2. Boehm, Hans-J. "Threads Cannot Be Implemented As a Library", ACM 2005, p265

3. Joshua Bloch "Effective Java, Second Edition", p. 283
4. Brian Goetz et al. Java Concurrency in Practice, 2006 pp348
5. [1] (http://www.javaconcurrencyinpractice.com/listings.html)
6. [2] (https://mailman.cs.umd.edu/mailman/private/javamemorymodel-discussion/2010-July/000422.html) Javamemorymodel-discussion mailing list
7. http://msdn.microsoft.com/en-us/library/12a04hfd(VS.100).aspx

外部链接

- Issues with the double checked locking mechanism captured in Jeu George's Blogs (http://purevirtuals.com/blog/2006/06/16/son-of-a-bug/) Pure Virtuals (http://purevirtuals.com/blog/2006/06/16/son-of-a-bug/)
- Implementation of Various Singleton Patterns including the Double Checked Locking Mechanism (http://www.tekpool.com/node/2693) at TEKPOOL (http://www.tekpool.com/?p=35)
- "Double Checked Locking" Description from the Portland Pattern Repository
- "Double Checked Locking is Broken" Description from the Portland Pattern Repository
- Paper "C++ and the Perils of Double-Checked Locking (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)" (475 KB) by Scott Meyers and Andrei Alexandrescu
- Article "Double-checked locking: Clever, but broken (http://www.javaworld.com/jw-02-2001/jw-0209-double.html)" by Brian Goetz
- The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html); David Bacon et al.
- Double-checked locking and the Singleton pattern (http://www-106.ibm.com/developerworks/java/library/j-dcl.html)
- Singleton Pattern and Thread Safety (http://www.oaklib.org/docs/oak/singleton.html)
- volatile keyword in VC++ 2005 (http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx)
- Java Examples and timing of double check locking solutions (http://blogs.sun.com/cwebster/entry/double_check_locking)
