深入理解高并发编程

(第1版)

写在前面

> 在 **冰河技术** 微信公众号中的【高并发】专题，更新了不少文章，有些读者反馈说，在公众号中刷历史文章不太方便，有时会忘记自己看到哪一篇了，当打开一篇文章时，似乎之前已经看过了，但就是不知道具体该看哪一篇了。相信很多小伙伴都会有这样的问题。那怎么办呢？最好的解决方案就是我把这些文章整理成PDF电子书，免费分享给大家，这样，小伙伴们看起来就方便多了。今天，我就将 **冰河技术** 微信公众号【高并发】专题中的文章，整理成《深入理解高并发编程（第1版）》 分享给大家，希望这本电子书能够给大家带来实质性的帮助。后续，我也会持续在 **冰河技术** 微信公众号中更新【高并发】专题，如果这本电子书能够给你带来帮助，请关注 **冰河技术** 微信公众号，我们一起进阶，一起牛逼。

关于作者

大数据架构师，Mykit系列开源框架作者，多年来致力于分布式系统架构、微服务、分布式数据库、分布式事务与大数据技术的研究，曾主导过众多分布式系统、微服务及大数据项目的架构设计、研发和实施落地。在高并发、高可用、高可扩展性、高可维护性和大数据等领域拥有丰富的架构经验。对Hadoop，Storm，Spark，Flink等大数据框架源码进行过深度分析，并具有丰富的实战经验。目前在研究云原生。公众号【冰河技术】作者，《海量数据处理与大数据技术实战》、《MySQL技术大全：开发、优化与运维实战》作者，基于最终消息可靠性的开源分布式事务框架mykit-transaction-message作者。

微信搜一搜

冰河技术

打开“微信 / 发现 / 搜一搜”搜索

源码分析篇

程序员究竟要不要读源码？

很多人觉得读源码比较枯燥，确实，读源码是要比看那些表面教你如何使用的文章要枯燥的多，也比不上刷抖音和微博来的轻松愉快。但是，读源码是一名程序员突破自我瓶颈，获得高薪和升职加薪的一个有效途径。通过阅读优秀的开源框架的源码，我们能够领略到框架作者设计框架的思维和思路，从中学习优秀的架构设计和代码设计。这些都是在那些只告诉你如何使用的文章中所学不到的，就更别提是刷抖音和微博了。

当你只停留在业务层面的CRUD开发而不思进取时，工作几年之后，你会发现你几乎除了使用啥都不会！此时，你在职场其实是毫无竞争优势的。你所反反复复做的工作对于刚入行的毕业生来说，给他们3个月时间，他们就能熟练上手。而你，反反复复做了几年的CRUD，没啥改变。对于企业来说，他们更加愿意雇佣那些成本低廉的新手，而不愿雇佣你！为啥？因为你给企业产出的价值未必比新入行的新手高，而你为企业带来的成本却远远高于新手！看到这里，知道为啥你工作几年后，想跳槽时，面试一个月薪几万+的职位，却只能仰望叹气了吧！！而比你工作年限少的人，却能够轻松面试比你薪资高出好几倍的职位！！不是他们运气好，而是他们比你掌握了更加深入的技能！！

当你在几年的工作时间里做的都是CRUD时，其实你的工作经验只有3个月；当你在3个月里，充分为自己规划好，在掌握基础业务开发的同时，抽时间为自己充电，掌握一些更加深入的技能，则你的工作经验会高于那些混迹职场几年的CRUD人员。

在职场还有一个现象，就是在某些企业会有一些不断加班疯狂撸代码的人，不是公司压榨员工，就是员工本身能力不行。当然，公司开发人员比较少，项目时间短的情况可以除外。往往那些疯狂加班撸代码的都是长期的CRUD者，他们干的比谁都累，拿的比谁都少。往往那些掌握了深入技能的人，看似很轻松，但是他们单位时间产出的价值远远高于CRUD人员疯狂撸一天代码产出的价值，因为那些CRUD人员一天下来产出的Bug，需要三天时间进行修正！！！

其实在职场，对于每个人非常重要的技能就是提升自己的核心竞争力，让自己变得更加有价值。

希望能够唤起你对知识的渴望。记住：工作年限并不等于工作经验！！！

线程与线程池

线程与多线程

1.线程

在操作系统中，线程是比进程更小的能够独立运行的基本单位。同时，它也是CPU调度的基本单位。线程本身基本上不拥有系统资源，只是拥有一些在运行时需要用到的系统资源，例如程序计数器，寄存器和栈等。一个进程中的所有线程可以共享进程中的所有资源。

2.多线程

多线程可以理解为在同一个程序中能够同时运行多个不同的线程来执行不同的任务，这些线程可以同时利用CPU的多个核心运行。多线程编程能够最大限度的利用CPU的资源。如果某一个线程的处理不需要占用CPU资源时（例如IO线程），可以使当前线程让出CPU资源来让其他线程能够获取到CPU资源，进而能够执行其他线程对应的任务，达到最大化利用CPU资源的目的。

线程的实现方式

在Java中，实现线程的方式大体上分为三种，通过继承Thread类、实现Runnable接口，实现Callable接口。简单的示例代码分别如下所示。

1.继承Thread类代码

```
package io.binghe.concurrent.executor.test;
/**
 * @author binghe
 * @version 1.0.0
 * @description 继承Thread实现线程
 */
public class ThreadTest extends Thread {
    @Override
    public void run() {
        //TODO 在此写在线程中执行的业务逻辑
    }
}
```

2.实现Runnable接口的代码

```java
package io.binghe.concurrent.executor.test;
/**
 * @author binghe
 * @version 1.0.0
 * @description 实现Runnable实现线程
 */
public class RunnableTest implements Runnable {
    @Override
    public void run() {
        //TODO 在此写在线程中执行的业务逻辑
    }
}
```

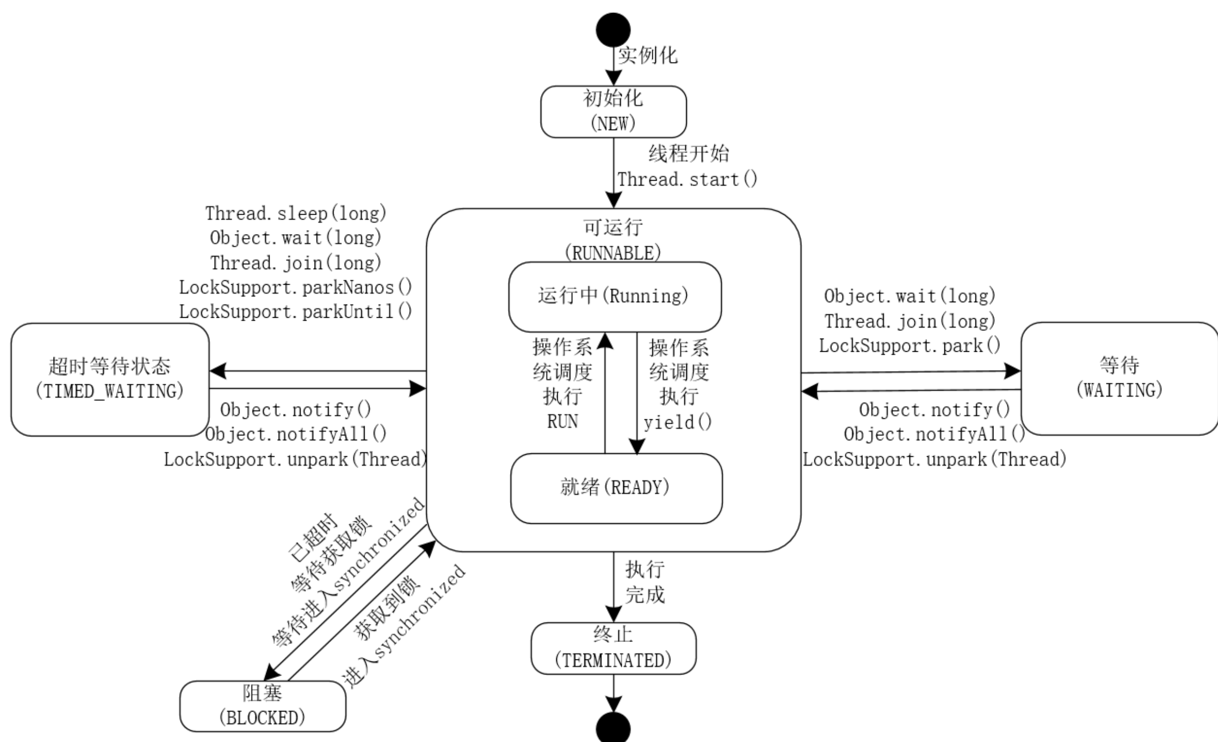
3.实现Callable接口的代码

```java
package io.binghe.concurrent.executor.test;

import java.util.concurrent.Callable;

/**
 * @author binghe
 * @version 1.0.0
 * @description 实现Callable实现线程
 */
public class CallableTest implements Callable<String> {
    @Override
    public String call() throws Exception {
        //TODO 在此写在线程中执行的业务逻辑
        return null;
    }
}
```

线程的生命周期

1.生命周期

一个线程从创建，到最终的消亡，需要经历多种不同的状态，而这些不同的线程状态，由始至终也构成了线程生命周期的不同阶段。线程的生命周期可以总结为下图。



其中，几个重要的状态如下所示。

- NEW：初始状态，线程被构建，但是还没有调用start()方法。
- RUNNABLE：可运行状态，可运行状态可以包括：运行中状态和就绪状态。

- BLOCKED：阻塞状态，处于这个状态的线程需要等待其他线程释放锁或者等待进入synchronized。
- WAITING：表示等待状态，处于该状态的线程需要等待其他线程对其进行通知或中断等操作，进而进入下一个状态。
- TIME_WAITING：超时等待状态。可以在一定的时间自行返回。
- TERMINATED：终止状态，当前线程执行完毕。

2.代码示例

为了更好的理解线程的生命周期，以及生命周期中的各个状态，接下来使用代码示例来输出线程的每个状态信息。

- WaitingTime

创建WaitingTime类，在while(true)循环中调用TimeUnit.SECONDS.sleep(long)方法来验证线程的TIMED_WARTING状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程不断休眠
 */
public class WaitingTime implements Runnable{
    @Override
    public void run() {
        while (true){
            waitSecond(200);
        }
    }
    //线程等待多少秒
    public static final void waitSecond(long seconds){
        try {
            TimeUnit.SECONDS.sleep(seconds);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- WaitingState

创建WaitingState类，此线程会在一个while(true)循环中，获取当前类Class对象的synchronized锁，也就是说，这个类无论创建多少个实例，synchronized锁都是同一个，并且线程会处于等待状态。接下来，在synchronized中使用当前类的Class对象的wait()方法，来验证线程的WAITING状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;
/**
 * @author binghe
 * @version 1.0.0
 * @description 线程在Warting上等待
 */
public class WaitingState implements Runnable {
    @Override
    public void run() {
        while (true){
            synchronized (WaitingState.class){
                try {
                    WaitingState.class.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- BlockedThread

BlockedThread主要是在synchronized代码块中的while(true)循环中调用TimeUnit.SECONDS.sleep(long)方法来验证线程的BLOCKED状态。当启动两个BlockedThread线程时，首先启动的线程会处于TIMED_WAITING状态，后启动的线程会处于BLOCKED状态。代码如下所示。

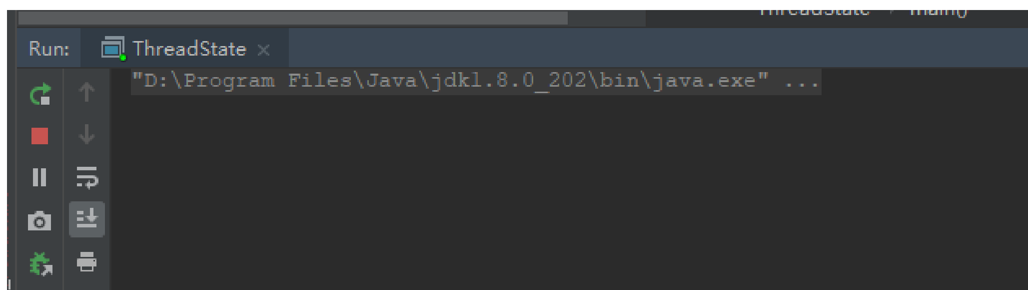
```
package io.binghe.concurrent.executor.state;
/**
 * @author binghe
 * @version 1.0.0
 * @description 加锁后不再释放锁
 */
public class BlockedThread implements Runnable {
    @Override
    public void run() {
        synchronized (BlockedThread.class){
            while (true){
                WaitingTime.waitSecond(100);
            }
        }
    }
}
```

- ThreadState

启动各个线程，验证各个线程输出的状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的各种状态，测试线程的生命周期
 */
public class ThreadState {

    public static void main(String[] args){
        new Thread(new WaitingTime(), "WaitingTimeThread").start();
        new Thread(new WaitingState(), "WaitingStateThread").start();

        //BlockedThread-01线程会抢到锁，BlockedThread-02线程会阻塞
        new Thread(new BlockedThread(), "BlockedThread-01").start();
        new Thread(new BlockedThread(), "BlockedThread-02").start();
    }
}
```

运行ThreadState类，如下所示。



可以看到，未输出任何结果信息。可以在命令行输入“jps”命令来查看运行的Java进程。

```
c:\>jps
21584 Jps
17828 KotlinCompileDaemon
12284 Launcher
24572
28492 ThreadState
```

可以看到ThreadSate进程的进程号为28492，接下来，输入“jstack 28492”来查看ThreadSate进程栈的信息，如下所示。

```
c:\>jstack 28492
2020-02-15 00:27:08
```

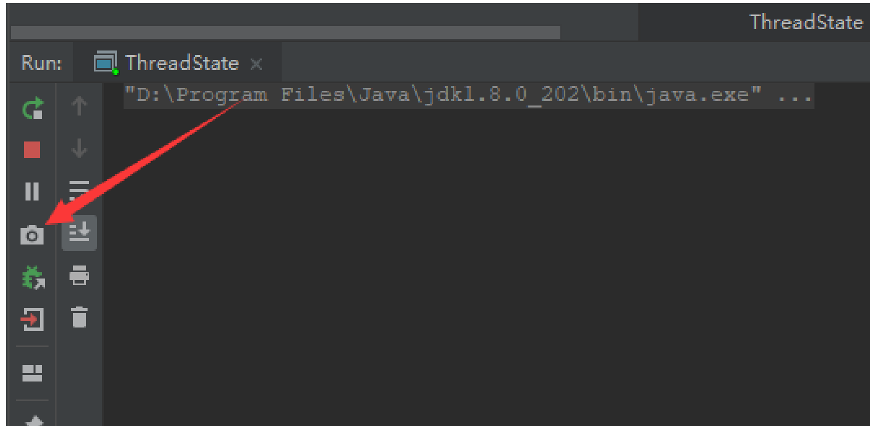
```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.202-b08 mixed mode):

"DestroyJavaVM" #16 prio=5 os_prio=0 tid=0x000000001ca05000 nid=0x1a4 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"BlockedThread-02" #15 prio=5 os_prio=0 tid=0x000000001ca04800 nid=0x6eb0 waiting for monitor entry
[0x000000001da4f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at io.binghe.concurrent.executor.state.BlockedThread.run(BlockedThread.java:28)
        - waiting to lock <0x0000000780a7e4e8> (a java.lang.Class for
io.binghe.concurrent.executor.state.BlockedThread)
        at java.lang.Thread.run(Thread.java:748)

"BlockedThread-01" #14 prio=5 os_prio=0 tid=0x000000001ca01800 nid=0x6e28 waiting on condition
[0x000000001d94f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:340)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
        at io.binghe.concurrent.executor.state.WaitingTime.waitSecond(WaitingTime.java:36)
        at io.binghe.concurrent.executor.state.BlockedThread.run(BlockedThread.java:28)
        - locked <0x0000000780a7e4e8> (a java.lang.Class for
io.binghe.concurrent.executor.state.BlockedThread)
        at java.lang.Thread.run(Thread.java:748)

"WaitingStateThread" #13 prio=5 os_prio=0 tid=0x000000001ca06000 nid=0x6fe4 in Object.wait()
[0x000000001d84f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780a7b488> (a java.lang.Class for
io.binghe.concurrent.executor.state.WaitingState)
        at java.lang.Object.wait(Object.java:502)
        at io.binghe.concurrent.executor.state.WaitingState.run(WaitingState.java:29)
        - locked <0x0000000780a7b488> (a java.lang.Class for
io.binghe.concurrent.executor.state.WaitingState)
        at java.lang.Thread.run(Thread.java:748)

"WaitingTimeThread" #12 prio=5 os_prio=0 tid=0x000000001c9f8800 nid=0x3858 waiting on condition
[0x000000001d74f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:340)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
        at io.binghe.concurrent.executor.state.WaitingTime.waitSecond(WaitingTime.java:36)
        at io.binghe.concurrent.executor.state.WaitingTime.run(WaitingTime.java:29)
        at java.lang.Thread.run(Thread.java:748)

"Service Thread" #11 daemon prio=9 os_prio=0 tid=0x000000001c935000 nid=0x6864 runnable
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread3" #10 daemon prio=9 os_prio=2 tid=0x000000001c88c800 nid=0x6a28 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread2" #9 daemon prio=9 os_prio=2 tid=0x000000001c880000 nid=0x6498 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os_prio=2 tid=0x000000001c87c000 nid=0x693c waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os_prio=2 tid=0x000000001c87b800 nid=0x5d00 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0 tid=0x000000001c862000 nid=0x6034 runnable
[0x000000001d04e000]
   java.lang.Thread.State: RUNNABLE
```

```
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
        at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
        at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
        - locked <0x0000000780b2fd88> (a java.io.InputStreamReader)
        at java.io.InputStreamReader.read(InputStreamReader.java:184)
        at java.io.BufferedReader.fill(BufferedReader.java:161)
        at java.io.BufferedReader.readLine(BufferedReader.java:324)
        - locked <0x0000000780b2fd88> (a java.io.InputStreamReader)
        at java.io.BufferedReader.readLine(BufferedReader.java:389)
        at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:64)

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x000000001c788800 nid=0x6794 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x000000001c7e3800 nid=0x3354 runnable
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001c771000 nid=0x6968 in Object.wait()
[0x000000001cd4f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780908ed0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:144)
        - locked <0x0000000780908ed0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:165)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:216)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000001c770800 nid=0x6590 in Object.wait()
[0x000000001cc4f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780906bf8> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
        - locked <0x0000000780906bf8> (a java.lang.ref.Reference$Lock)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"VM Thread" os_prio=2 tid=0x000000001a979800 nid=0x5c2c runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00000000033b9000 nid=0x4dc0 runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00000000033ba800 nid=0x6690 runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00000000033bc000 nid=0x30b0 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00000000033be800 nid=0x6f68 runnable

"GC task thread#4 (ParallelGC)" os_prio=0 tid=0x00000000033c1000 nid=0x6478 runnable

"GC task thread#5 (ParallelGC)" os_prio=0 tid=0x00000000033c2000 nid=0x4fe4 runnable

"GC task thread#6 (ParallelGC)" os_prio=0 tid=0x00000000033c5000 nid=0x584 runnable

"GC task thread#7 (ParallelGC)" os_prio=0 tid=0x00000000033c6800 nid=0x6988 runnable

"VM Periodic Task Thread" os_prio=2 tid=0x000000001c959800 nid=0x645c waiting on condition

JNI global references: 12
```

由以上输出的信息可以看出：名称为WaitingTimeThread的线程处于TIMED_WAITING状态；名称为WaitingStateThread的线程处于WAITING状态；名称为BlockedThread-01的线程处于TIMED_WAITING状态；名称为BlockedThread-02的线程处于BLOCKED状态。

注意：使用jps结合jstack命令可以分析线上生产环境的Java进程的异常信息。

也可以直接点击IDEA下图所示的图表直接打印出线程的堆栈信息。



输出的结果信息与使用"jstack 进程号"命令输出的信息基本一致。

线程的执行顺序

线程的执行顺序是不确定的

调用Thread的start()方法启动线程时，线程的执行顺序是不确定的。也就是说，在同一个方法中，连续创建多个线程后，调用线程的start()方法的顺序并不能决定线程的执行顺序。

例如，这里，看一个简单的示例程序，如下所示。

```
package io.binghe.concurrent.lab03;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的顺序，直接调用Thread.start()方法执行不能确保线程的执行顺序
 */
public class ThreadSort01 {
    public static void main(String[] args){
        Thread thread1 = new Thread(() -> {
            System.out.println("thread1");
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("thread2");
        });
        Thread thread3 = new Thread(() -> {
            System.out.println("thread3");
        });

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

在ThreadSort01类中分别创建了三个不同的线程，thread1、thread2和thread3，接下来，在程序中按照顺序分别调用thread1.start()、thread2.start()和thread3.start()方法来分别启动三个不同的线程。

那么，问题来了，线程的执行顺序是否按照thread1、thread2和thread3的顺序执行呢？运行ThreadSort01的main方法，结果如下所示。

```
thread1
thread2
thread3
```

再次运行时，结果如下所示。

```
thread1
thread3
thread2
```

第三次运行时，结果如下所示。

```
thread2
thread3
thread1
```

注意：每个人运行的情况可能都不一样。

可以看到，每次运行程序时，线程的执行顺序可能不同。线程的启动顺序并不能决定线程的执行顺序。

如何确保线程的执行顺序

1.确保线程执行顺序的简单示例

在实际业务场景中，有时，后启动的线程可能需要依赖先启动的线程执行完成才能正确的执行线程中的业务逻辑。此时，就需要确保线程的执行顺序。那么如何确保线程的执行顺序呢？

可以使用Thread类中的join()方法来确保线程的执行顺序。例如，下面的测试代码。

```
package io.binghe.concurrent.lab03;
/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的顺序，Thread.join()方法能够确保线程的执行顺序
 */
public class ThreadSort02 {
    public static void main(String[] args) throws InterruptedException {

        Thread thread1 = new Thread(() -> {
            System.out.println("thread1");
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("thread2");
        });
        Thread thread3 = new Thread(() -> {
            System.out.println("thread3");
        });

        thread1.start();

        //实际上让主线程等待子线程执行完成
        thread1.join();

        thread2.start();
        thread2.join();

        thread3.start();
        thread3.join();
    }
}
```

可以看到，ThreadSort02类比ThreadSort01类，在每个线程的启动方法下面添加了调用线程的join()方法。此时，运行ThreadSort02类，结果如下所示。

```
thread1
thread2
thread3
```

再次运行时，结果如下所示。

```
thread1
thread2
thread3
```

第三次运行时，结果如下所示。

```
thread1
thread2
thread3
```

可以看到，每次运行的结果都是相同的，所以，使用Thread的join()方法能够保证线程的先后执行顺序。

2.join方法如何确保线程的执行顺序

既然Thread类的join()方法能够确保线程的执行顺序，我们就一起来看看Thread类的join()方法到底是个什么鬼。

进入Thread的join()方法，如下所示。

```
public final void join() throws InterruptedException {
    join(0);
}
```

可以看到join()方法调用同类中的一个有参join()方法，并传递参数0。继续跟进代码，如下所示。

```
public final synchronized void join(long millis)
throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}
```

可以看到，有一个long类型参数的join()方法使用了synchroinzed修饰，说明这个方法同一时刻只能被一个实例或者方法调用。由于，传递的参数为0，所以，程序会进入如下代码逻辑。

```
if (millis == 0) {
    while (isAlive()) {
        wait(0);
    }
}
```

首先，在代码中以while循环的方式来判断当前线程是否已经启动处于活跃状态，如果已经启动处于活跃状态，则调用同类中的wait()方法，并传递参数0。继续跟进wait()方法，如下所示。

```
public final native void wait(long timeout) throws InterruptedException;
```

可以看到，wait()方法是一个本地方法，通过JNI的方式调用JDK底层的方法来使线程等待执行完成。

需要注意的是，调用线程的wait()方法时，会使主线程处于等待状态，等待子线程执行完成后再次向下执行。也就是说，在ThreadSort02类的main()方法中，调用子线程的join()方法，会阻塞main()方法的执行，当子线程执行完成后，main()方法会继续向下执行，启动第二个子线程，并执行子线程的业务逻辑，以此类推。

Java中的Callable和Future

在Java的多线程编程中，除了Thread类和Runnable接口外，不得不说的就是Callable接口Future接口了。使用继承Thread类或者实现Runnable接口的线程，无法返回最终的执行结果数据，只能等待线程执行完成。此时，如果想要获取线程执行后的返回结果，那么，Callable和Future就派上用场了。

Callable接口

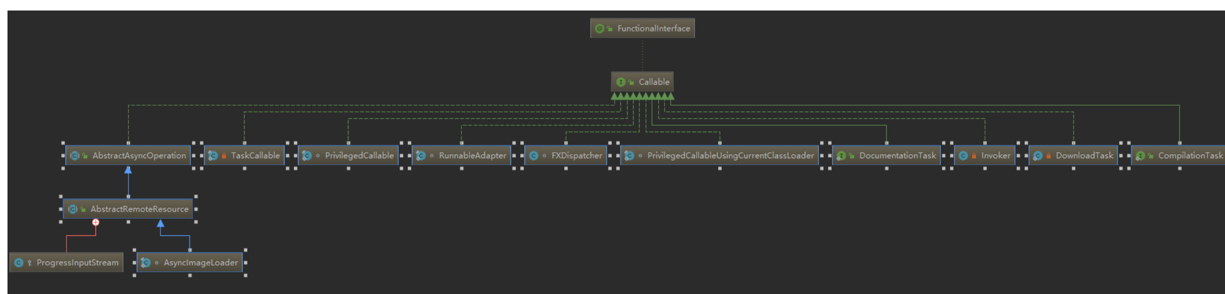
1.Callable接口介绍

Callable接口是JDK1.5新增的泛型接口，在JDK1.8中，被声明为函数式接口，如下所示。

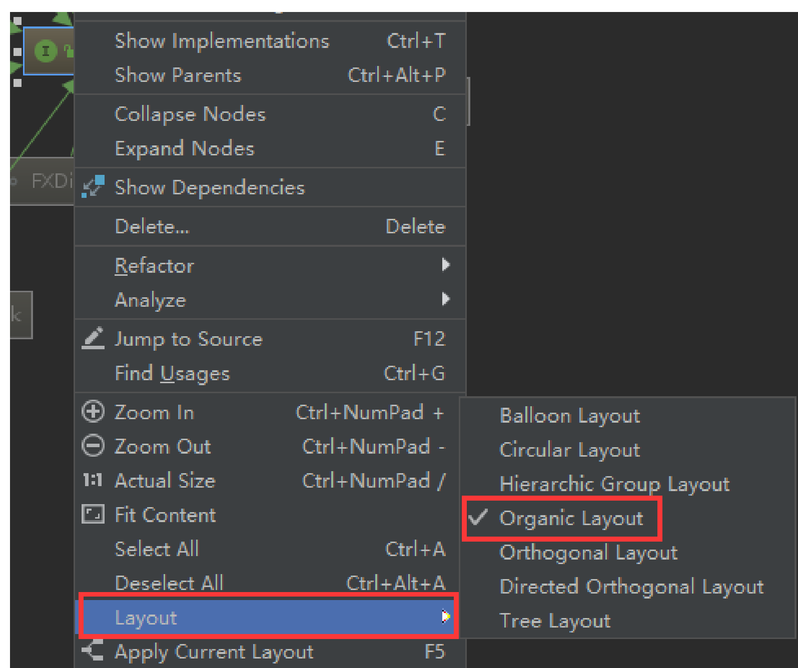
```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

在JDK 1.8中只声明有一个方法的接口为函数式接口，函数式接口可以使用@FunctionalInterface注解修饰，也可以不使用@FunctionalInterface注解修饰。只要一个接口中只包含有一个方法，那么，这个接口就是函数式接口。

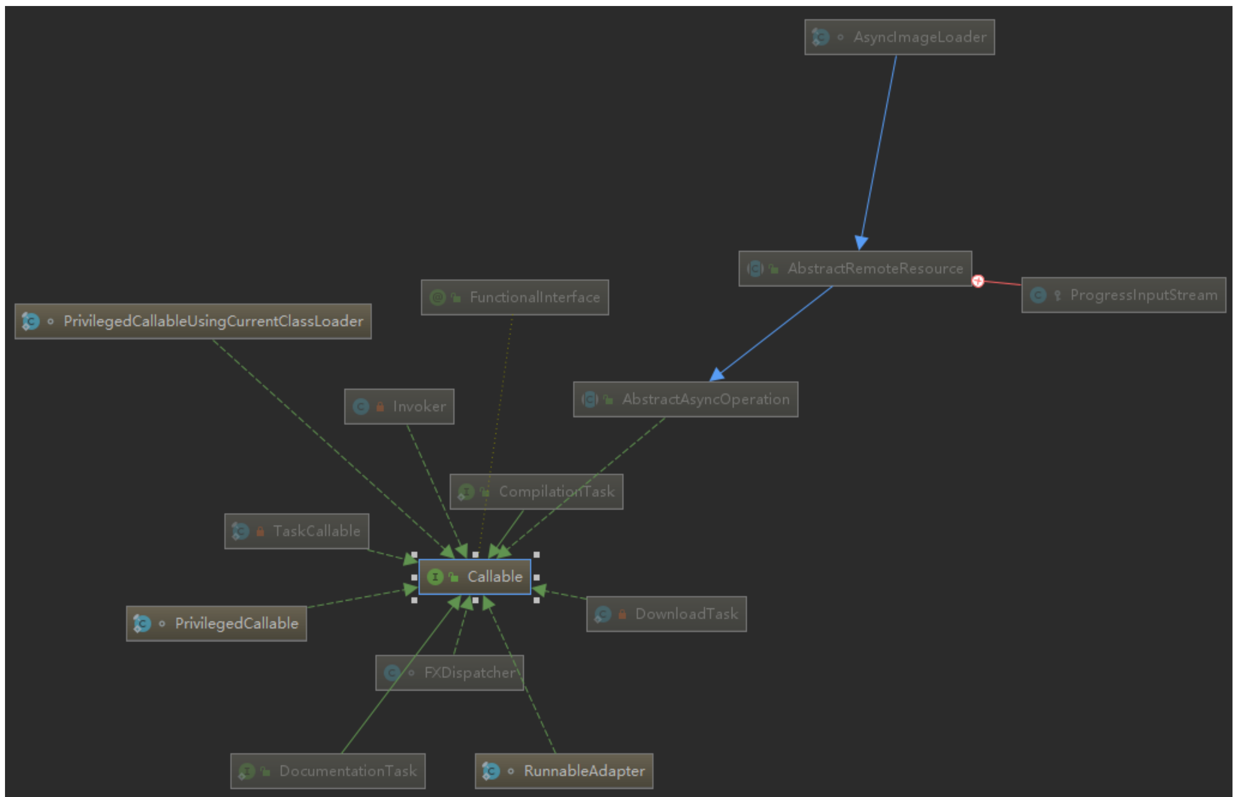
在JDK中，实现Callable接口的子类如下图所示。



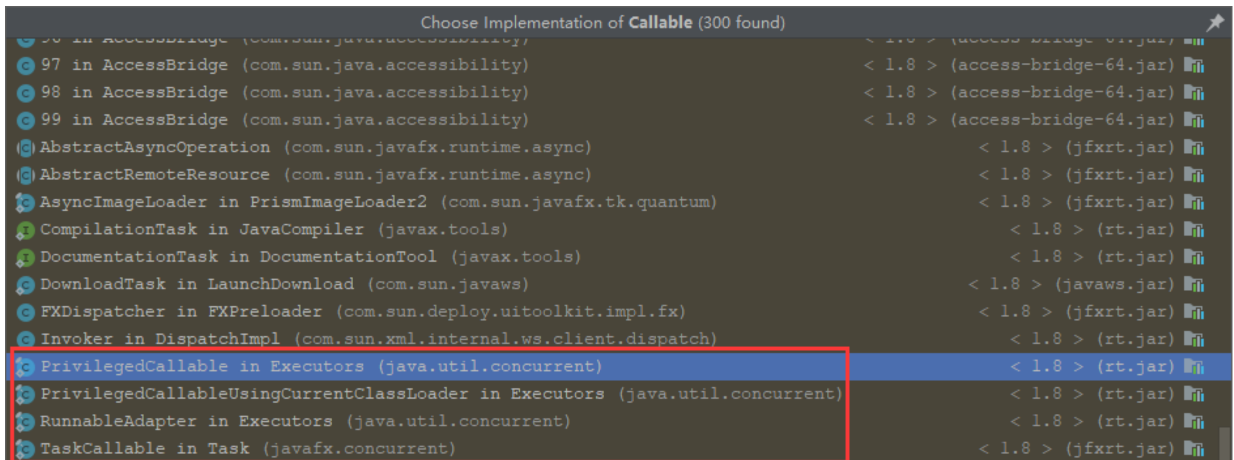
默认的子类层级关系图看不清，这里，可以通过IDEA右键Callable接口，选择“Layout”来指定Callable接口的实现类图的不同结构，如下所示。



这里，可以选择“Organic Layout”选项，选择后的Callable接口的子类的结构如下图所示。



在实现Callable接口的子类中，有几个比较重要的类，如下图所示。



分别是：Executors类中的静态内部类：PrivilegedCallable、PrivilegedCallableUsingCurrentClassLoader、RunnableAdapter和Task类下的TaskCallable。

2.实现Callable接口的重要类分析

接下来，分析的类主要有：PrivilegedCallable、PrivilegedCallableUsingCurrentClassLoader、RunnableAdapter和Task类下的TaskCallable。虽然这些类在实际工作中很少被直接用到，但是作为一名合格的开发工程师，设置是秃顶的资深专家来说，了解并掌握这些类的实现有助你进一步理解Callable接口，并提高专业技能（头发再掉一批，哇哈哈哈。。。）。

- PrivilegedCallable

PrivilegedCallable类是Callable接口的一个特殊实现类，它表明Callable对象有某种特权来访问系统的某种资源，PrivilegedCallable类的源代码如下所示。

```
/**
 * A callable that runs under established access control settings
 */
static final class PrivilegedCallable<T> implements Callable<T> {
    private final Callable<T> task;
    private final AccessControlContext acc;

    PrivilegedCallable(Callable<T> task) {
        this.task = task;
        this.acc = AccessController.getContext();
    }
```

```java
    public T call() throws Exception {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<T>() {
                    public T run() throws Exception {
                        return task.call();
                    }
                }, acc);
        } catch (PrivilegedActionException e) {
            throw e.getException();
        }
    }
}
```

从PrivilegedCallable类的源代码来看，可以将PrivilegedCallable看成是对Callable接口的封装，并且这个类也继承了Callable接口。

在PrivilegedCallable类中有两个成员变量，分别是Callable接口的实例对象和AccessControlContext类的实例对象，如下所示。

```java
private final Callable<T> task;
private final AccessControlContext acc;
```

其中，AccessControlContext类可以理解为一个具有系统资源访问决策的上下文类，通过这个类可以访问系统的特定资源。通过类的构造方法可以看出，在实例化AccessControlContext类的对象时，只需要传递Callable接口子类的对象即可，如下所示。

```java
PrivilegedCallable(Callable<T> task) {
    this.task = task;
    this.acc = AccessController.getContext();
}
```

AccessControlContext类的对象是通过AccessController类的getContext()方法获取的，这里，查看AccessController类的getContext()方法，如下所示。

```java
public static AccessControlContext getContext(){
    AccessControlContext acc = getStackAccessControlContext();
    if (acc == null) {
        return new AccessControlContext(null, true);
    } else {
        return acc.optimize();
    }
}
```

通过AccessController的getContext()方法可以看出，首先通过getStackAccessControlContext()方法来获取AccessControlContext对象实例。如果获取的AccessControlContext对象实例为空，则通过调用AccessControlContext类的构造方法实例化，否则，调用AccessControlContext对象实例的optimize()方法返回AccessControlContext对象实例。

这里，我们先看下getStackAccessControlContext()方法是个什么鬼。

```java
private static native AccessControlContext getStackAccessControlContext();
```

原来是个本地方法，方法的字面意思就是获取能够访问系统栈的决策上下文对象。

接下来，我们回到PrivilegedCallable类的call()方法，如下所示。

```java
public T call() throws Exception {
    try {
        return AccessController.doPrivileged(
            new PrivilegedExceptionAction<T>() {
                public T run() throws Exception {
                    return task.call();
                }
            }, acc);
    } catch (PrivilegedActionException e) {
        throw e.getException();
    }
}
```

通过调用AccessController.doPrivileged()方法，传递PrivilegedExceptionAction。接口对象和AccessControlContext对象，并最终返回泛型的实例对象。

首先，看下AccessController.doPrivileged()方法，如下所示。

```
@CallerSensitive
public static native <T> T
    doPrivileged(PrivilegedExceptionAction<T> action,
                 AccessControlContext context)
    throws PrivilegedActionException;
```

可以看到，又是一个本地方法。也就是说，最终的执行情况是将PrivilegedExceptionAction接口对象和AccessControlContext对象实例传递给这个本地方法执行。并且在PrivilegedExceptionAction接口对象的run()方法中调用Callable接口的call()方法来执行最终的业务逻辑，并且返回泛型对象。

- PrivilegedCallableUsingCurrentClassLoader

此类表示为在已经建立的特定访问控制和当前的类加载器下运行的Callable类，源代码如下所示。

```
/**
 * A callable that runs under established access control settings and
 * current ClassLoader
 */
static final class PrivilegedCallableUsingCurrentClassLoader<T> implements Callable<T> {
    private final Callable<T> task;
    private final AccessControlContext acc;
    private final ClassLoader ccl;

    PrivilegedCallableUsingCurrentClassLoader(Callable<T> task) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(SecurityConstants.GET_CLASSLOADER_PERMISSION);
            sm.checkPermission(new RuntimePermission("setContextClassLoader"));
        }
        this.task = task;
        this.acc = AccessController.getContext();
        this.ccl = Thread.currentThread().getContextClassLoader();
    }

    public T call() throws Exception {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<T>() {
                    public T run() throws Exception {
                        Thread t = Thread.currentThread();
                        ClassLoader cl = t.getContextClassLoader();
                        if (ccl == cl) {
                            return task.call();
                        } else {
                            t.setContextClassLoader(ccl);
                            try {
                                return task.call();
                            } finally {
                                t.setContextClassLoader(cl);
                            }
                        }
                    }
                }, acc);
        } catch (PrivilegedActionException e) {
            throw e.getException();
        }
    }
}
```

这个类理解起来比较简单，首先，在类中定义了三个成员变量，如下所示。

```
private final Callable<T> task;
private final AccessControlContext acc;
private final ClassLoader ccl;
```

接下来，通过构造方法注入Callable对象，在构造方法中，首先获取系统安全管理器对象实例，通过系统安全管理器对象实例检查是否具有获取ClassLoader和设置ContextClassLoader的权限。并在构造方法中为三个成员变量赋值，如下所示。

```
PrivilegedCallableUsingCurrentClassLoader(Callable<T> task) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(SecurityConstants.GET_CLASSLOADER_PERMISSION);
        sm.checkPermission(new RuntimePermission("setContextClassLoader"));
    }
    this.task = task;
    this.acc = AccessController.getContext();
    this.ccl = Thread.currentThread().getContextClassLoader();
}
```

接下来，通过调用call()方法来执行具体的业务逻辑，如下所示。

```
public T call() throws Exception {
    try {
        return AccessController.doPrivileged(
            new PrivilegedExceptionAction<T>() {
                public T run() throws Exception {
                    Thread t = Thread.currentThread();
                    ClassLoader cl = t.getContextClassLoader();
                    if (ccl == cl) {
                        return task.call();
                    } else {
                        t.setContextClassLoader(ccl);
                        try {
                            return task.call();
                        } finally {
                            t.setContextClassLoader(cl);
                        }
                    }
                }
            }, acc);
    } catch (PrivilegedActionException e) {
        throw e.getException();
    }
}
```

在call()方法中同样是通过调用AccessController类的本地方法doPrivileged，传递PrivilegedExceptionAction接口的实例对象和AccessControlContext类的对象实例。

具体执行逻辑为：在PrivilegedExceptionAction对象的run()方法中获取当前线程的ContextClassLoader对象，如果在构造方法中获取的ClassLoader对象与此处的ContextClassLoader对象是同一个对象（不止对象实例相同，而且内存地址也相同），则直接调用Callable对象的call()方法返回结果。否则，将PrivilegedExceptionAction对象的run()方法中的当前线程的ContextClassLoader设置为在构造方法中获取的类加载器对象，接下来，再调用Callable对象的call()方法返回结果。最终将当前线程的ContextClassLoader重置为之前的ContextClassLoader。

- RunnableAdapter

RunnableAdapter类比较简单，给定运行的任务和结果，运行给定的任务并返回给定的结果，源代码如下所示。

```
/**
 * A callable that runs given task and returns given result
 */
static final class RunnableAdapter<T> implements Callable<T> {
    final Runnable task;
    final T result;
    RunnableAdapter(Runnable task, T result) {
        this.task = task;
        this.result = result;
    }
    public T call() {
        task.run();
        return result;
    }
}
```

- TaskCallable

TaskCallable类是javafx.concurrent.Task类的静态内部类，TaskCallable类主要是实现了Callable接口并且被定义为FutureTask的类，并且在这个类中允许我们拦截call()方法来更新task任务的状态。源代码如下所示。

```
private static final class TaskCallable<V> implements Callable<V> {

    private Task<V> task;
    private TaskCallable() { }

    @Override
    public V call() throws Exception {
        task.started = true;
        task.runLater(() -> {
            task.setState(State.SCHEDULED);
            task.setState(State.RUNNING);
        });
        try {
            final V result = task.call();
            if (!task.isCancelled()) {
                task.runLater(() -> {
                    task.updateValue(result);
                    task.setState(State.SUCCEEDED);
                });
                return result;
            } else {
                return null;
            }
        } catch (final Throwable th) {
            task.runLater(() -> {
                task._setException(th);
                task.setState(State.FAILED);
            });
            if (th instanceof Exception) {
                throw (Exception) th;
            } else {
                throw new Exception(th);
            }
        }
    }
}
```

从TaskCallable类的源代码可以看出，只定义了一个Task类型的成员变量。下面主要分析TaskCallable类的call()方法。

当程序的执行进入到call()方法时，首先将task对象的started属性设置为true，表示任务已经开始，并且将任务的状态依次设置为State.SCHEDULED和State.RUNNING，依次触发任务的调度事件和运行事件。如下所示。

```
task.started = true;
task.runLater(() -> {
    task.setState(State.SCHEDULED);
    task.setState(State.RUNNING);
});
```

接下来，在try代码块中执行Task对象的call()方法，返回泛型对象。如果任务没有被取消，则更新任务的缓存，将调用call()方法返回的泛型对象绑定到Task对象中的ObjectProperty对象中，其中，ObjectProperty在Task类中的定义如下。

```
private final ObjectProperty<V> value = new SimpleObjectProperty<>(this, "value");
```

接下来，将任务的状态设置为成功状态。如下所示。

```
try {
    final V result = task.call();
    if (!task.isCancelled()) {
        task.runLater(() -> {
            task.updateValue(result);
            task.setState(State.SUCCEEDED);
        });
        return result;
    } else {
        return null;
    }
}
```

如果程序抛出了异常或者错误，会进入catch()代码块，设置Task对象的Exception信息并将状态设置为State.FAILED，也就是将任务标记为失败。接下来，判断异常或错误的类型，如果是Exception类型的异常，则直接强转为Exception类型的异常并抛出。否则，将异常或者错误封装为Exception对象并抛出，如下所示。

```
catch (final Throwable th) {
    task.runLater(() -> {
        task._setException(th);
        task.setState(State.FAILED);
    });
    if (th instanceof Exception) {
        throw (Exception) th;
    } else {
        throw new Exception(th);
    }
}
```

两种异步模型与深度解析Future接口

两种异步模型

在Java的并发编程中，大体上会分为两种异步编程模型，一类是直接以异步的形式来并行运行其他的任务，不需要返回任务的结果数据。一类是以异步的形式运行其他任务，需要返回结果。

1.无返回结果的异步模型

无返回结果的异步任务，可以直接将任务丢进线程或线程池中运行，此时，无法直接获得任务的执行结果数据，一种方式是可以使用回调方法来获取任务的运行结果。

具体的方案是：定义一个回调接口，并在接口中定义接收任务结果数据的方法，具体逻辑在回调接口的实现类中完成。将回调接口与任务参数一同放进线程或线程池中运行，任务运行后调用接口方法，执行回调接口实现类中的逻辑来处理结果数据。这里，给出一个简单的示例供参考。

- 定义回调接口

```
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 定义回调接口
 */
public interface TaskCallable<T> {
    T callable(T t);
}
```

便于接口的通用型，这里为回调接口定义了泛型。

- 定义任务结果数据的封装类

```
package io.binghe.concurrent.lab04;

import java.io.Serializable;

/**
 * @author binghe
 * @version 1.0.0
```

```java
 * @description 任务执行结果
 */
public class TaskResult implements Serializable {
    private static final long serialVersionUID = 8678277072402730062L;
    /**
     * 任务状态
     */
    private Integer taskStatus;

    /**
     * 任务消息
     */
    private String taskMessage;

    /**
     * 任务结果数据
     */
    private String taskResult;

    //省略getter和setter方法
    @Override
    public String toString() {
        return "TaskResult{" +
                "taskStatus=" + taskStatus +
                ", taskMessage='" + taskMessage + '\'' +
                ", taskResult='" + taskResult + '\'' +
                '}';
    }
}
```

- 创建回调接口的实现类

回调接口的实现类主要用来对任务的返回结果进行相应的业务处理，这里，为了方便演示，只是将结果数据返回。大家需要根据具体的业务场景来做相应的分析和处理。

```java
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 回调函数的实现类
 */
public class TaskHandler implements TaskCallable<TaskResult> {
    @Override
public TaskResult callable(TaskResult taskResult) {
//TODO 拿到结果数据后进一步处理
    System.out.println(taskResult.toString());
        return taskResult;
    }
}
```

- 创建任务的执行类

任务的执行类是具体执行任务的类，实现Runnable接口，在此类中定义一个回调接口类型的成员变量和一个String类型的任务参数（模拟任务的参数），并在构造方法中注入回调接口和任务参数。在run方法中执行任务，任务完成后将任务的结果数据封装成TaskResult对象，调用回调接口的方法将TaskResult对象传递到回调方法中。

```java
package io.binghe.concurrent.lab04;
/**
 * @author binghe
 * @version 1.0.0
 * @description 任务执行类
 */
public class TaskExecutor implements Runnable{
    private TaskCallable<TaskResult> taskCallable;
    private String taskParameter;

    public TaskExecutor(TaskCallable<TaskResult> taskCallable, String taskParameter){
        this.taskCallable = taskCallable;
        this.taskParameter = taskParameter;
```

```java
    }

    @Override
    public void run() {
        //TODO 一系列业务逻辑,将结果数据封装成TaskResult对象并返回
        TaskResult result = new TaskResult();
        result.setTaskStatus(1);
        result.setTaskMessage(this.taskParameter);
        result.setTaskResult("异步回调成功");
        taskCallable.callable(result);
    }
}
```

到这里，整个大的框架算是完成了，接下来，就是测试看能否获取到异步任务的结果了。

- 异步任务测试类

```java
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试回调
 */
public class TaskCallableTest {
    public static void main(String[] args){
        TaskCallable<TaskResult> taskCallable = new TaskHandler();
        TaskExecutor taskExecutor = new TaskExecutor(taskCallable, "测试回调任务");
        new Thread(taskExecutor).start();
    }
}
```

在测试类中，使用Thread类创建一个新的线程，并启动线程运行任务。运行程序最终的接口数据如下所示。

```
TaskResult{taskStatus=1, taskMessage='测试回调任务', taskResult='异步回调成功'}
```

大家可以细细品味下这种获取异步结果的方式。这里，只是简单的使用了Thread类来创建并启动线程，也可以使用线程池的方式实现。大家可自行实现以线程池的方式通过回调接口获取异步结果。

2.有返回结果的异步模型

尽管使用回调接口能够获取异步任务的结果，但是这种方式使用起来略显复杂。在JDK中提供了可以直接返回异步结果的处理方案。最常用的就是使用Future接口或者其实现类FutureTask来接收任务的返回结果。

- 使用Future接口获取异步结果

使用Future接口往往配合线程池来获取异步执行结果，如下所示。

```java
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试Future获取异步结果
 */
public class FutureTest {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<String> future = executorService.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "测试Future获取异步结果";
            }
        });
        System.out.println(future.get());
        executorService.shutdown();
    }
```

```
}
```

运行结果如下所示。

```
测试Future获取异步结果
```

- 使用FutureTask类获取异步结果

FutureTask类既可以结合Thread类使用也可以结合线程池使用，接下来，就看下这两种使用方式。

结合Thread类的使用示例如下所示。

```java
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试FutureTask获取异步结果
 */
public class FutureTaskTest {

    public static void main(String[] args)throws ExecutionException, InterruptedException{
        FutureTask<String> futureTask = new FutureTask<>(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "测试FutureTask获取异步结果";
            }
        });
        new Thread(futureTask).start();
        System.out.println(futureTask.get());
    }
}
```

运行结果如下所示。

```
测试FutureTask获取异步结果
```

结合线程池的使用示例如下。

```java
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试FutureTask获取异步结果
 */
public class FutureTaskTest {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        FutureTask<String> futureTask = new FutureTask<>(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "测试FutureTask获取异步结果";
            }
        });
        executorService.execute(futureTask);
        System.out.println(futureTask.get());
        executorService.shutdown();
    }
}
```

运行结果如下所示。

```
测试FutureTask获取异步结果
```

可以看到使用Future接口或者FutureTask类来获取异步结果比使用回调接口获取异步结果简单多了。注意：实现异步的方式很多，这里只是用多线程举例。

接下来，就深入分析下Future接口。

深度解析Future接口

1.Future接口

Future是JDK1.5新增的异步编程接口，其源代码如下所示。

```
package java.util.concurrent;

public interface Future<V> {

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

可以看到，在Future接口中，总共定义了5个抽象方法。接下来，就分别介绍下这5个方法的含义。

- cancel(boolean)

取消任务的执行，接收一个boolean类型的参数，成功取消任务，则返回true，否则返回false。当任务已经完成，已经结束或者因其他原因不能取消时，方法会返回false，表示任务取消失败。当任务未启动调用了此方法，并且结果返回true（取消成功），则当前任务不再运行。如果任务已经启动，会根据当前传递的boolean类型的参数来决定是否中断当前运行的线程来取消当前运行的任务。

- isCancelled()

判断任务在完成之前是否被取消，如果在任务完成之前被取消，则返回true；否则，返回false。

这里需要注意一个细节：只有任务未启动，或者在完成之前被取消，才会返回true，表示任务已经被成功取消。其他情况都会返回false。

- isDone()

判断任务是否已经完成，如果任务正常结束、抛出异常退出、被取消，都会返回true，表示任务已经完成。

- get()

当任务完成时，直接返回任务的结果数据；当任务未完成时，等待任务完成并返回任务的结果数据。

- get(long, TimeUnit)

当任务完成时，直接返回任务的结果数据；当任务未完成时，等待任务完成，并设置了超时等待时间。在超时时间内任务完成，则返回结果；否则，抛出TimeoutException异常。

2.RunnableFuture接口

Future接口有一个重要的子接口，那就是RunnableFuture接口，RunnableFuture接口不但继承了Future接口，而且继承了java.lang.Runnable接口，其源代码如下所示。

```
package java.util.concurrent;

public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

这里，问一下，RunnableFuture接口中有几个抽象方法？想好了再说！哈哈哈。。。

这个接口比较简单run()方法就是运行任务时调用的方法。

3.FutureTask类

FutureTask类是RunnableFuture接口的一个非常重要的实现类，它实现了RunnableFuture接口、Future接口和Runnable接口的所有方法。FutureTask类的源代码比较多，这个就不粘贴了，大家自行到java.util.concurrent下查看。

（1）FutureTask类中的变量与常量

在FutureTask类中首先定义了一个状态变量state，这个变量使用了volatile关键字修饰，这里，大家只需要知道volatile关键字通过内存屏障和禁止重排序优化来实现线程安全，后续会单独深度分析volatile关键字是如何保证线程安全的。紧接着，定义了几个任务运行时的状态常量，如下所示。

```
private volatile int state;
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

其中，代码注释中给出了几个可能的状态变更流程，如下所示。

```
NEW -> COMPLETING -> NORMAL
NEW -> COMPLETING -> EXCEPTIONAL
NEW -> CANCELLED
NEW -> INTERRUPTING -> INTERRUPTED
```

接下来，定义了其他几个成员变量，如下所示。

```
private Callable<V> callable;
private Object outcome;
private volatile Thread runner;
private volatile WaitNode waiters;
```

又看到我们所熟悉的Callable接口了，Callable接口那肯定就是用来调用call()方法执行具体任务了。

- outcome：Object类型，表示通过get()方法获取到的结果数据或者异常信息。
- runner：运行Callable的线程，运行期间会使用CAS保证线程安全，这里大家只需要知道CAS是Java保证线程安全的一种方式，后续文章中会深度分析CAS如何保证线程安全。
- waiters：WaitNode类型的变量，表示等待线程的堆栈，在FutureTask的实现中，会通过CAS结合此堆栈交换任务的运行状态。

看一下WaitNode类的定义，如下所示。

```
static final class WaitNode {
    volatile Thread thread;
    volatile WaitNode next;
    WaitNode() { thread = Thread.currentThread(); }
}
```

可以看到，WaitNode类是FutureTask类的静态内部类，类中定义了一个Thread成员变量和指向下一个WaitNode节点的引用。其中通过构造方法将thread变量设置为当前线程。

（2）构造方法

接下来，是FutureTask的两个构造方法，比较简单，如下所示。

```
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;
}

public FutureTask(Runnable runnable, V result) {
    this.callable = Executors.callable(runnable, result);
    this.state = NEW;
}
```

（3）是否取消与完成方法

继续向下看源码，看到一个任务是否取消的方法，和一个任务是否完成的方法，如下所示。

```
public boolean isCancelled() {
    return state >= CANCELLED;
}

public boolean isDone() {
    return state != NEW;
}
```

这两方法中，都是通过判断任务的状态来判定任务是否已取消和已完成的。为啥会这样判断呢？再次查看FutureTask类中定义的状态常量发现，其常量的定义是有规律的，并不是随意定义的。其中，大于或者等于CANCELLED的常量为CANCELLED、INTERRUPTING和INTERRUPTED，这三个状态均可以表示线程已经被取消。当状态不等于NEW时，可以表示任务已经完成。

通过这里，大家可以学到一点：以后在编码过程中，要按照规律来定义自己使用的状态，尤其是涉及到业务中有频繁的状态变更的操作，有规律的状态可使业务处理变得事半功倍，这也是通过看别人的源码设计能够学到的，这里，建议大家还是多看别人写的优秀的开源框架的源码。

（4）取消方法

我们继续向下看源码，接下来，看到的是cancel(boolean)方法，如下所示。

```
public boolean cancel(boolean mayInterruptIfRunning) {
    if (!(state == NEW &&
          UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
              mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
        return false;
    try {    // in case call to interrupt throws exception
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
            }
        }
    } finally {
        finishCompletion();
    }
    return true;
}
```

接下来，拆解cancel(boolean)方法。在cancel(boolean)方法中，首先判断任务的状态和CAS的操作结果，如果任务的状态不等于NEW或者CAS的操作返回false，则直接返回false，表示任务取消失败。如下所示。

```
if (!(state == NEW &&
      UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
          mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
    return false;
```

接下来，在try代码块中，首先判断是否可以中断当前任务所在的线程来取消任务的运行。如果可以中断当前任务所在的线程，则以一个Thread临时变量来指向运行任务的线程，当指向的变量不为空时，调用线程对象的interrupt()方法来中断线程的运行，最后将线程标记为被中断的状态。如下所示。

```
try {
    if (mayInterruptIfRunning) {
        try {
            Thread t = runner;
            if (t != null)
                t.interrupt();
        } finally { // final state
            UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
        }
    }
}
```

这里，发现变更任务状态使用的是UNSAFE.putOrderedInt()方法，这个方法是个什么鬼呢？点进去看一下，如下所示。

```
public native void putOrderedInt(Object var1, long var2, int var4);
```

可以看到，又是一个本地方法，嘿嘿，这里先不管它，后续文章会详解这些方法的作用。

接下来，cancel(boolean)方法会进入finally代码块，如下所示。

```
finally {
    finishCompletion();
}
```

可以看到在finallly代码块中调用了finishCompletion()方法，顾名思义，finishCompletion()方法表示结束任务的运行，接下来看看它是如何实现的。点到finishCompletion()方法中看一下，如下所示。

```
private void finishCompletion() {
    // assert state > COMPLETING;
    for (WaitNode q; (q = waiters) != null;) {
        if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
            for (;;) {
                Thread t = q.thread;
                if (t != null) {
                    q.thread = null;
                    LockSupport.unpark(t);
                }
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }
    done();
    callable = null;        // to reduce footprint
}
```

在finishCompletion()方法中，首先定义一个for循环，循环终止因子为waiters为null，在循环中，判断CAS操作是否成功，如果成功进行if条件中的逻辑。首先，定义一个for自旋循环，在自旋循环体中，唤醒WaitNode堆栈中的线程，使其运行完成。当WaitNode堆栈中的线程运行完成后，通过break退出外层for循环。接下来调用done()方法。done()方法又是个什么鬼呢？点进去看一下，如下所示。

```
protected void done() { }
```

可以看到，done()方法是一个空的方法体，交由子类来实现具体的业务逻辑。

当我们的具体业务中，需要在取消任务时，执行一些额外的业务逻辑，可以在子类中覆写done()方法的实现。

（5）get()方法

继续向下看FutureTask类的代码，FutureTask类中实现了两个get()方法，如下所示。

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);
    return report(s);
}

public V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException {
    if (unit == null)
        throw new NullPointerException();
    int s = state;
    if (s <= COMPLETING &&
        (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)
        throw new TimeoutException();
    return report(s);
```

```
}
```

没参数的get()方法为当任务未运行完成时，会阻塞，直到返回任务结果。有参数的get()方法为当任务未运行完成，并且等待时间超出了超时时间，会TimeoutException异常。

两个get()方法的主要逻辑差不多，一个没有超时设置，一个有超时设置，这里说一下主要逻辑。判断任务的当前状态是否小于或者等于COMPLETING，也就是说，任务是NEW状态或者COMPLETING，调用awaitDone()方法，看下awaitDone()方法的实现，如下所示。

```
private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }

        int s = state;
        if (s > COMPLETING) {
            if (q != null)
                q.thread = null;
            return s;
        }
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();
        else if (q == null)
            q = new WaitNode();
        else if (!queued)
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                 q.next = waiters, q);
        else if (timed) {
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) {
                removeWaiter(q);
                return state;
            }
            LockSupport.parkNanos(this, nanos);
        }
        else
            LockSupport.park(this);
    }
}
```

接下来，拆解awaitDone()方法。在awaitDone()方法中，最重要的就是for自旋循环，在循环中首先判断当前线程是否被中断，如果已经被中断，则调用removeWaiter()将当前线程从堆栈中移除，并且抛出InterruptedException异常，如下所示。

```
if (Thread.interrupted()) {
    removeWaiter(q);
    throw new InterruptedException();
}
```

接下来，判断任务的当前状态是否完成，如果完成，并且堆栈句柄不为空，则将堆栈中的当前线程设置为空，返回当前任务的状态，如下所示。

```
int s = state;
if (s > COMPLETING) {
    if (q != null)
        q.thread = null;
    return s;
}
```

当任务的状态为COMPLETING时，使当前线程让出CPU资源，如下所示。

```
else if (s == COMPLETING)
    Thread.yield();
```

如果堆栈为空，则创建堆栈对象，如下所示。

```
else if (q == null)
    q = new WaitNode();
```

如果queued变量为false，通过CAS操作为queued赋值，如果awaitDone()方法传递的timed参数为true，则计算超时时间，当时间已超时，则在堆栈中移除当前线程并返回任务状态，如下所示。如果未超时，则重置超时时间，如下所示。

```
else if (!queued)
    queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                         q.next = waiters, q);
else if (timed) {
    nanos = deadline - System.nanoTime();
    if (nanos <= 0L) {
        removeWaiter(q);
        return state;
    }
    LockSupport.parkNanos(this, nanos);
}
```

如果不满足上述的所有条件，则将当前线程设置为等待状态，如下所示。

```
else
    LockSupport.park(this);
```

接下来，回到get()方法中，当awaitDone()方法返回结果，或者任务的状态不满足条件时，都会调用report()方法，并将当前任务的状态传递到report()方法中，并返回结果，如下所示。

```
return report(s);
```

看来，这里还要看下report()方法啊，点进去看下report()方法的实现，如下所示。

```
private V report(int s) throws ExecutionException {
    Object x = outcome;
    if (s == NORMAL)
        return (V)x;
    if (s >= CANCELLED)
        throw new CancellationException();
    throw new ExecutionException((Throwable)x);
}
```

可以看到，report()方法的实现比较简单，首先，将outcome数据赋值给x变量，接下来，主要是判断接收到的任务状态，如果状态为NORMAL，则将x强转为泛型类型返回；当任务的状态大于或者等于CANCELLED，也就是任务已经取消，则抛出CancellationException异常，其他情况则抛出ExecutionException异常。

至此，get()方法分析完成。注意：一定要理解get()方法的实现，因为get()方法是我们使用Future接口和FutureTask类时，使用的比较频繁的一个方法。

（6）set()方法与setException()方法

继续看FutureTask类的代码，接下来看到的是set()方法与setException()方法，如下所示。

```
protected void set(V v) {
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = v;
        UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
        finishCompletion();
    }
}

protected void setException(Throwable t) {
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = t;
        UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state
        finishCompletion();
    }
}
```

通过源码可以看出，set()方法与setException()方法整体逻辑几乎一样，只是在设置任务状态时一个将状态设置为NORMAL，一个将状态设置为EXCEPTIONAL。

至于finishCompletion()方法，前面已经分析过。

（7）run()方法与runAndReset()方法

接下来，就是run()方法了，run()方法的源代码如下所示。

```
public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                set(result);
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
}
```

可以这么说，只要使用了Future和FutureTask，就必然会调用run()方法来运行任务，掌握run()方法的流程是非常有必要的。在run()方法中，如果当前状态不是NEW，或者CAS操作返回的结果为false，则直接返回，不再执行后续逻辑，如下所示。

```
if (state != NEW ||
    !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                 null, Thread.currentThread()))
    return;
```

接下来，在try代码块中，将成员变量callable赋值给一个临时变量c，判断临时变量不等于null，并且任务状态为NEW，则调用Callable接口的call()方法，并接收结果数据。并将ran变量设置为true。当程序抛出异常时，将接收结果的变量设置为null，ran变量设置为false，并且调用setException()方法将任务的状态设置为EXCEPTIONA。接下来，如果ran变量为true，则调用set()方法，如下所示。

```
try {
    Callable<V> c = callable;
    if (c != null && state == NEW) {
        V result;
        boolean ran;
        try {
            result = c.call();
            ran = true;
        } catch (Throwable ex) {
            result = null;
            ran = false;
            setException(ex);
        }
        if (ran)
```

```
            set(result);
        }
    }
```

接下来，程序会进入finally代码块中，如下所示。

```
finally {
    // runner must be non-null until state is settled to
    // prevent concurrent calls to run()
    runner = null;
    // state must be re-read after nulling runner to prevent
    // leaked interrupts
    int s = state;
    if (s >= INTERRUPTING)
        handlePossibleCancellationInterrupt(s);
}
```

这里，将runner设置为null，如果任务的当前状态大于或者等于INTERRUPTING，也就是线程被中断了。则调用handlePossibleCancellationInterrupt()方法，接下来，看下handlePossibleCancellationInterrupt()方法的实现。

```
private void handlePossibleCancellationInterrupt(int s) {
    if (s == INTERRUPTING)
        while (state == INTERRUPTING)
            Thread.yield();
}
```

可以看到，handlePossibleCancellationInterrupt()方法的实现比较简单，当任务的状态为INTERRUPTING时，使用while()循环，条件为当前任务状态为INTERRUPTING，将当前线程占用的CPU资源释放，也就是说，当任务运行完成后，释放线程所占用的资源。

runAndReset()方法的逻辑与run()差不多，只是runAndReset()方法会在finally代码块中将任务状态重置为NEW。runAndReset()方法的源代码如下所示，就不重复说了。

```
protected boolean runAndReset() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return false;
    boolean ran = false;
    int s = state;
    try {
        Callable<V> c = callable;
        if (c != null && s == NEW) {
            try {
                c.call(); // don't set result
                ran = true;
            } catch (Throwable ex) {
                setException(ex);
            }
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
    return ran && s == NEW;
}
```

（8）removeWaiter()方法

removeWaiter()方法中主要是使用自旋循环的方式来移除WaitNode中的线程，比较简单，如下所示。

```
private void removeWaiter(WaitNode node) {
    if (node != null) {
        node.thread = null;
```

```java
        retry:
        for (;;) {          // restart on removeWaiter race
            for (WaitNode pred = null, q = waiters, s; q != null; q = s) {
                s = q.next;
                if (q.thread != null)
                    pred = q;
                else if (pred != null) {
                    pred.next = s;
                    if (pred.thread == null) // check for race
                        continue retry;
                }
                else if (!UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                      q, s))
                    continue retry;
            }
            break;
        }
    }
}
```

最后，在FutureTask类的最后，有如下代码。

```java
// Unsafe mechanics
private static final sun.misc.Unsafe UNSAFE;
private static final long stateOffset;
private static final long runnerOffset;
private static final long waitersOffset;
static {
    try {
        UNSAFE = sun.misc.Unsafe.getUnsafe();
        Class<?> k = FutureTask.class;
        stateOffset = UNSAFE.objectFieldOffset
            (k.getDeclaredField("state"));
        runnerOffset = UNSAFE.objectFieldOffset
            (k.getDeclaredField("runner"));
        waitersOffset = UNSAFE.objectFieldOffset
            (k.getDeclaredField("waiters"));
    } catch (Exception e) {
        throw new Error(e);
    }
}
```

关于这些代码的作用，会在后续深度解析CAS文章中详细说明，这里就不再探讨。

至此，关于Future接口和FutureTask类的源码就分析完了。

SimpleDateFormat类的线程安全问题

提起SimpleDateFormat类，想必做过Java开发的童鞋都不会感到陌生。没错，它就是Java中提供的日期时间的转化类。这里，为什么说SimpleDateFormat类有线程安全问题呢？有些小伙伴可能会提出疑问：我们生产环境上一直在使用SimpleDateFormat类来解析和格式化日期和时间类型的数据，一直都没有问题啊！我的回答是：没错，那是因为你们的系统达不到SimpleDateFormat类出现问题的并发量，也就是说你们的系统没啥负载！

接下来，我们就一起看下在高并发下SimpleDateFormat类为何会出现安全问题，以及如何解决SimpleDateFormat类的安全问题。

重现SimpleDateFormat类的线程安全问题

为了重现SimpleDateFormat类的线程安全问题，一种比较简单的方式就是使用线程池结合Java并发包中的CountDownLatch类和Semaphore类来重现线程安全问题。

有关CountDownLatch类和Semaphore类的具体用法和底层原理与源码解析在【高并发专题】后文会深度分析。这里，大家只需要知道CountDownLatch类可以使一个线程等待其他线程各自执行完毕后再执行。而Semaphore类可以理解为一个计数信号量，必须由获取它的线程释放，经常用来限制访问某些资源的线程数量，例如限流等。

好了，先来看下重现SimpleDateFormat类的线程安全问题的代码，如下所示。

```java
package io.binghe.concurrent.lab06;
```

```java
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试SimpleDateFormat的线程不安全问题
 */
public class SimpleDateFormatTest01 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

可以看到，在SimpleDateFormatTest01类中，首先定义了两个常量，一个是程序执行的总次数，一个是同时运行的线程数量。程序中结合线程池和CountDownLatch类与Semaphore类来模拟高并发的业务场景。其中，有关日期转化的代码只有如下一行。

```java
simpleDateFormat.parse("2020-01-01");
```

当程序捕获到异常时，打印相关的信息，并退出整个程序的运行。当程序正确运行后，会打印“所有线程格式化日期成功”。

运行程序输出的结果信息如下所示。

```
Exception in thread "pool-1-thread-4" Exception in thread "pool-1-thread-1" Exception in thread "pool-
1-thread-2" 线程：pool-1-thread-7 格式化日期失败
线程：pool-1-thread-9 格式化日期失败
线程：pool-1-thread-10 格式化日期失败
```

```
Exception in thread "pool-1-thread-3" Exception in thread "pool-1-thread-5" Exception in thread "pool-1-thread-6" 线程：pool-1-thread-15 格式化日期失败
线程：pool-1-thread-21 格式化日期失败
Exception in thread "pool-1-thread-23" 线程：pool-1-thread-16 格式化日期失败
线程：pool-1-thread-11 格式化日期失败
java.lang.ArrayIndexOutOfBoundsException
线程：pool-1-thread-27 格式化日期失败
	at java.lang.System.arraycopy(Native Method)
	at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:597)
	at java.lang.StringBuffer.append(StringBuffer.java:367)
	at java.text.DigitList.getLong(DigitList.java:191)线程：pool-1-thread-25 格式化日期失败

	at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
	at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
	at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
线程：pool-1-thread-14 格式化日期失败
	at java.text.DateFormat.parse(DateFormat.java:364)
	at io.binghe.concurrent.lab06.SimpleDateFormatTest01.lambda$main$0(SimpleDateFormatTest01.java:47)
线程：pool-1-thread-13 格式化日期失败 at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)

	at java.lang.Thread.run(Thread.java:748)
java.lang.NumberFormatException: For input string: ""
	at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
线程：pool-1-thread-20 格式化日期失败 at java.lang.Long.parseLong(Long.java:601)
	at java.lang.Long.parseLong(Long.java:631)

	at java.text.DigitList.getLong(DigitList.java:195)
	at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
	at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:2162)
	at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
	at java.text.DateFormat.parse(DateFormat.java:364)
	at io.binghe.concurrent.lab06.SimpleDateFormatTest01.lambda$main$0(SimpleDateFormatTest01.java:47)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
	at java.lang.Thread.run(Thread.java:748)
java.lang.NumberFormatException: For input string: ""
	at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
	at java.lang.Long.parseLong(Long.java:601)
	at java.lang.Long.parseLong(Long.java:631)
	at java.text.DigitList.getLong(DigitList.java:195)
	at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
	at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
	at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
	at java.text.DateFormat.parse(DateFormat.java:364)

Process finished with exit code 1
```

说明，在高并发下使用SimpleDateFormat类格式化日期时抛出了异常，SimpleDateFormat类不是线程安全的！！！

接下来，我们就看下，SimpleDateFormat类为何不是线程安全的。

SimpleDateFormat类为何不是线程安全的

那么，接下来，我们就一起来看看真正引起SimpleDateFormat类线程不安全的根本原因。

通过查看SimpleDateFormat类的源码，我们得知：SimpleDateFormat是继承自DateFormat类，DateFormat类中维护了一个全局的Calendar变量，如下所示。

```java
/**
 * The {@link Calendar} instance used for calculating the date-time fields
 * and the instant of time. This field is used for both formatting and
 * parsing.
 *
 * <p>Subclasses should initialize this field to a {@link Calendar}
 * appropriate for the {@link Locale} associated with this
 * <code>DateFormat</code>.
 * @serial
 */
protected Calendar calendar;
```

从注释可以看出，这个Calendar对象既用于格式化也用于解析日期时间。接下来，我们再查看parse()方法接近最后的部分。

```java
@Override
public Date parse(String text, ParsePosition pos){
    ################此处省略N行代码##################
    Date parsedDate;
    try {
        parsedDate = calb.establish(calendar).getTime();
        // If the year value is ambiguous,
        // then the two-digit year == the default start year
        if (ambiguousYear[0]) {
            if (parsedDate.before(defaultCenturyStart)) {
                parsedDate = calb.addYear(100).establish(calendar).getTime();
            }
        }
    }
    // An IllegalArgumentException will be thrown by Calendar.getTime()
    // if any fields are out of range, e.g., MONTH == 17.
    catch (IllegalArgumentException e) {
        pos.errorIndex = start;
        pos.index = oldStart;
        return null;
    }
    return parsedDate;
}
```

可见，最后的返回值是通过调用CalendarBuilder.establish()方法获得的，而这个方法的参数正好就是前面的Calendar对象。

接下来，我们再来看看CalendarBuilder.establish()方法，如下所示。

```java
Calendar establish(Calendar cal) {
    boolean weekDate = isSet(WEEK_YEAR)
        && field[WEEK_YEAR] > field[YEAR];
    if (weekDate && !cal.isWeekDateSupported()) {
        // Use YEAR instead
        if (!isSet(YEAR)) {
            set(YEAR, field[MAX_FIELD + WEEK_YEAR]);
        }
        weekDate = false;
    }

    cal.clear();
    // Set the fields from the min stamp to the max stamp so that
    // the field resolution works in the Calendar.
    for (int stamp = MINIMUM_USER_STAMP; stamp < nextStamp; stamp++) {
        for (int index = 0; index <= maxFieldIndex; index++) {
            if (field[index] == stamp) {
                cal.set(index, field[MAX_FIELD + index]);
                break;
            }
        }
    }

    if (weekDate) {
        int weekOfYear = isSet(WEEK_OF_YEAR) ? field[MAX_FIELD + WEEK_OF_YEAR] : 1;
        int dayOfWeek = isSet(DAY_OF_WEEK) ?
            field[MAX_FIELD + DAY_OF_WEEK] : cal.getFirstDayOfWeek();
```

```java
        if (!isValidDayOfWeek(dayOfWeek) && cal.isLenient()) {
            if (dayOfWeek >= 8) {
                dayOfWeek--;
                weekOfYear += dayOfWeek / 7;
                dayOfWeek = (dayOfWeek % 7) + 1;
            } else {
                while (dayOfWeek <= 0) {
                    dayOfWeek += 7;
                    weekOfYear--;
                }
            }
            dayOfWeek = toCalendarDayOfWeek(dayOfWeek);
        }
        cal.setWeekDate(field[MAX_FIELD + WEEK_YEAR], weekOfYear, dayOfWeek);
    }
    return cal;
}
```

在CalendarBuilder.establish()方法中先后调用了cal.clear()与cal.set()，也就是先清除cal对象中设置的值，再重新设置新的值。由于Calendar内部并没有线程安全机制，并且这两个操作也都不是原子性的，所以当多个线程同时操作一个SimpleDateFormat时就会引起cal的值混乱。类似地， **format()方法也存在同样的问题。**

因此， SimpleDateFormat类不是线程安全的根本原因是：**DateFormat类中的Calendar对象被多线程共享，而Calendar对象本身不支持线程安全。**

那么，得知了SimpleDateFormat类不是线程安全的，以及造成SimpleDateFormat类不是线程安全的原因，那么如何解决这个问题呢？接下来，我们就一起探讨下如何解决SimpleDateFormat类在高并发场景下的线程安全问题。

解决SimpleDateFormat类的线程安全问题

解决SimpleDateFormat类在高并发场景下的线程安全问题可以有多种方式，这里，就列举几个常用的方式供参考，大家也可以在评论区给出更多的解决方案。

1.局部变量法

最简单的一种方式就是将SimpleDateFormat类对象定义成局部变量，如下所示的代码，将SimpleDateFormat类对象定义在parse(String)方法的上面，即可解决问题。

```java
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 局部变量法解决SimpleDateFormat类的线程安全问题
 */
public class SimpleDateFormatTest02 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
```

```
                    System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失败");
                    e.printStackTrace();
                    System.exit(1);
                }catch (NumberFormatException e){
                    System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失败");
                    e.printStackTrace();
                    System.exit(1);
                }
                semaphore.release();
            } catch (InterruptedException e) {
                System.out.println("信号量发生错误");
                e.printStackTrace();
                System.exit(1);
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    System.out.println("所有线程格式化日期成功");
  }
}
```

此时运行修改后的程序，输出结果如下所示。

```
所有线程格式化日期成功
```

至于在高并发场景下使用局部变量为何能解决线程的安全问题，会在【JVM专题】的JVM内存模式相关内容中深入剖析，这里不做过多的介绍了。

当然，这种方式在高并发下会创建大量的SimpleDateFormat类对象，影响程序的性能，所以，这种方式在实际生产环境不太被推荐。

2.synchronized锁方式

将SimpleDateFormat类对象定义成全局静态变量，此时所有线程共享SimpleDateFormat类对象，此时在调用格式化时间的方法时，对SimpleDateFormat对象进行同步即可，代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过Synchronized锁解决SimpleDateFormat类的线程安全问题
 */
public class SimpleDateFormatTest03 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
```

```
                    try {
                        synchronized (simpleDateFormat){
                            simpleDateFormat.parse("2020-01-01");
                        }
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

此时，解决问题的关键代码如下所示。

```
synchronized (simpleDateFormat){
    simpleDateFormat.parse("2020-01-01");
}
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

需要注意的是，虽然这种方式能够解决SimpleDateFormat类的线程安全问题，但是由于在程序的执行过程中，为SimpleDateFormat类对象加上了synchronized锁，导致同一时刻只能有一个线程执行parse(String)方法。此时，会影响程序的执行性能，在要求高并发的生产环境下，此种方式也是不太推荐使用的。

3.Lock锁方式

Lock锁方式与synchronized锁方式实现原理相同，都是在高并发下通过JVM的锁机制来保证程序的线程安全。通过Lock锁方式解决问题的代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过Lock锁解决SimpleDateFormat类的线程安全问题
 */
public class SimpleDateFormatTest04 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
```

```java
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
    //Lock对象
    private static Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        lock.lock();
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }finally {
                        lock.unlock();
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

通过代码可以得知，首先，定义了一个Lock类型的全局静态变量作为加锁和释放锁的句柄。然后在simpleDateFormat.parse(String)代码之前通过lock.lock()加锁。这里需要注意的一点是：为防止程序抛出异常而导致锁不能被释放，一定要将释放锁的操作放到finally代码块中，如下所示。

```java
finally {
    lock.unlock();
}
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

此种方式同样会影响高并发场景下的性能，不太建议在高并发的生产环境使用。

4.ThreadLocal方式

使用ThreadLocal存储每个线程拥有的SimpleDateFormat对象的副本，能够有效的避免多线程造成的线程安全问题，使用ThreadLocal解决线程安全问题的代码如下所示。

```java
package io.binghe.concurrent.lab06;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
```

```java
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过ThreadLocal解决SimpleDateFormat类的线程安全问题
 */
public class SimpleDateFormatTest05 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static ThreadLocal<DateFormat> threadLocal = new ThreadLocal<DateFormat>(){
        @Override
        protected DateFormat initialValue() {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        threadLocal.get().parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

通过代码可以得知，将每个线程使用的SimpleDateFormat副本保存在ThreadLocal中，各个线程在使用时互不干扰，从而解决了线程安全问题。

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

此种方式运行效率比较高，推荐在高并发业务场景的生产环境使用。

另外，使用ThreadLocal也可以写成如下形式的代码，效果是一样的。

```java
package io.binghe.concurrent.lab06;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过ThreadLocal解决SimpleDateFormat类的线程安全问题
 */
public class SimpleDateFormatTest06 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static ThreadLocal<DateFormat> threadLocal = new ThreadLocal<DateFormat>();

    private static DateFormat getDateFormat(){
        DateFormat dateFormat = threadLocal.get();
        if(dateFormat == null){
            dateFormat = new SimpleDateFormat("yyyy-MM-dd");
            threadLocal.set(dateFormat);
        }
        return dateFormat;
    }

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        getDateFormat().parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

5.DateTimeFormatter方式

DateTimeFormatter是Java8提供的新的日期时间API中的类，DateTimeFormatter类是线程安全的，可以在高并发场景下直接使用DateTimeFormatter类来处理日期的格式化操作。代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过DateTimeFormatter类解决线程安全问题
 */
public class SimpleDateFormatTest07 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

   private static DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        LocalDate.parse("2020-01-01", formatter);
                    }catch (Exception e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

可以看到，DateTimeFormatter类是线程安全的，可以在高并发场景下直接使用DateTimeFormatter类来处理日期的格式化操作。

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

使用DateTimeFormatter类来处理日期的格式化操作运行效率比较高，推荐在高并发业务场景的生产环境使用。

6.joda-time方式

joda-time是第三方处理日期时间格式化的类库，是线程安全的。如果使用joda-time来处理日期和时间的格式化，则需要引入第三方类库。这里，我以Maven为例，如下所示引入joda-time库。

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.9.9</version>
</dependency>
```

引入joda-time库后，实现的程序代码如下所示。

```
package io.binghe.concurrent.lab06;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过DateTimeFormatter类解决线程安全问题
 */
public class SimpleDateFormatTest08 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static DateTimeFormatter dateTimeFormatter = DateTimeFormat.forPattern("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        DateTime.parse("2020-01-01", dateTimeFormatter).toDate();
                    }catch (Exception e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失
败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

这里，需要注意的是：DateTime类是org.joda.time包下的类，DateTimeFormat类和DateTimeFormatter类都是org.joda.time.format包下的类，如下所示。

```
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

使用joda-time库来处理日期的格式化操作运行效率比较高，推荐在高并发业务场景的生产环境使用。

综上所示：在解决解决SimpleDateFormat类的线程安全问题的几种方案中，局部变量法由于线程每次执行格式化时间时，都会创建SimpleDateFormat类的对象，这会导致创建大量的SimpleDateFormat对象，浪费运行空间和消耗服务器的性能，因为JVM创建和销毁对象是要耗费性能的。所以，不推荐在高并发要求的生产环境使用。

synchronized锁方式和Lock锁方式在处理问题的本质上是一致的，通过加锁的方式，使同一时刻只能有一个线程执行格式化日期和时间的操作。这种方式虽然减少了SimpleDateFormat对象的创建，但是由于同步锁的存在，导致性能下降，所以，不推荐在高并发要求的生产环境使用。

ThreadLocal通过保存各个线程的SimpleDateFormat类对象的副本，使每个线程在运行时，各自使用自身绑定的SimpleDateFormat对象，互不干扰，执行性能比较高，推荐在高并发的生产环境使用。

DateTimeFormatter是Java 8中提供的处理日期和时间的类，DateTimeFormatter类本身就是线程安全的，经压测，DateTimeFormatter类处理日期和时间的性能效果还不错（后文单独写一篇关于高并发下性能压测的文章）。所以，推荐在高并发场景下的生产环境使用。

joda-time是第三方处理日期和时间的类库，线程安全，性能经过高并发的考验，推荐在高并发场景下的生产环境使用。

深度解析ThreadPoolExecutor类源码

抛砖引玉

既然Java中支持以多线程的方式来执行相应的任务，但为什么在JDK1.5中又提供了线程池技术呢？这个问题大家自行脑补，多动脑，肯定没坏处，哈哈哈。。。

说起Java中的线程池技术，在很多框架和异步处理中间件中都有涉及，而且性能经受起了长久的考验。可以这样说，Java的线程池技术是Java最核心的技术之一，在Java的高并发领域中，Java的线程池技术是一个永远绕不开的话题。既然Java的线程池技术这么重要（怎么能说是这么重要呢？那是相当的重要，那家伙老重要了，哈哈哈），那么，本文我们就来简单的说下线程池与ThreadPoolExecutor类。至于线程池中的各个技术细节和ThreadPoolExecutor的底层原理和源码解析，我们会在【高并发专题】专栏中进行深度解析。

引言：本文是高并发中线程池的开篇之作，就暂时先不深入讲解，只是让大家从整体上认识下线程池中最核心的类之一——ThreadPoolExecutor，关于ThreadPoolExecutor的底层原理和源码实现，以及线程池中的其他技术细节的底层原理和源码实现，我们会在【高并发专题】接下来的文章中，进行死磕。

Thread直接创建线程的弊端

（1）每次new Thread新建对象，性能差。
（2）线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或OOM。
（3）缺少更多的功能，如更多执行、定期执行、线程中断。
（4）其他弊端，大家自行脑补，多动脑，没坏处，哈哈。

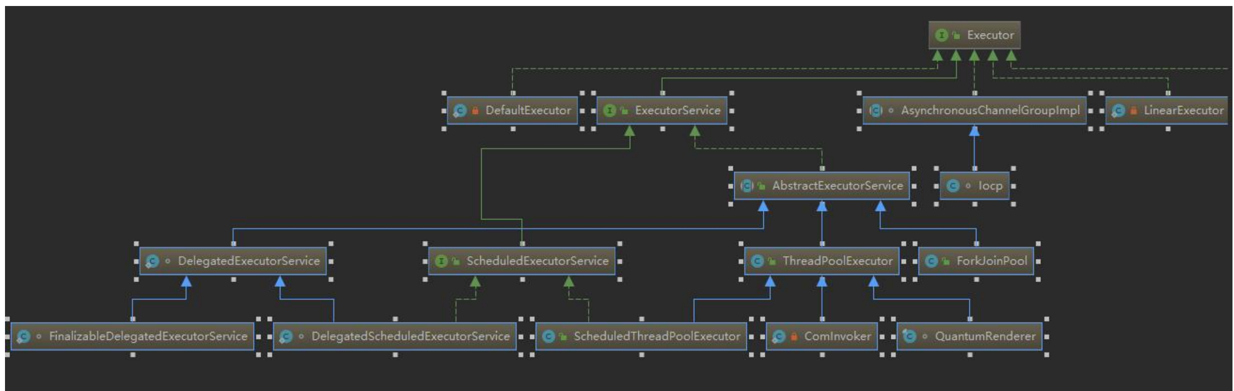
线程池的好处

（1）重用存在的线程，减少对象创建、消亡的开销，性能佳。
（2）可以有效控制最大并发线程数，提高系统资源利用率，同时可以避免过多资源竞争，避免阻塞。
（3）提供定时执行、定期执行、单线程、并发数控制等功能。
（4）提供支持线程池监控的方法，可对线程池的资源进行实时监控。
（5）其他好处，大家自行脑补，多动脑，没坏处，哈哈。

线程池

1.线程池类结构关系

线程池中的一些接口和类的结构关系如下图所示。



后文会死磕这些接口和类的底层原理和源码。

2.创建线程池常用的类——Executors

- Executors.newCachedThreadPool：创建一个可缓存的线程池，如果线程池的大小超过了需要，可以灵活回收空闲线程，如果没有可回收线程，则新建线程
- Executors.newFixedThreadPool：创建一个定长的线程池，可以控制线程的最大并发数，超出的线程会在队列中等待
- Executors.newScheduledThreadPool：创建一个定长的线程池，支持定时、周期性的任务执行
- Executors.newSingleThreadExecutor: 创建一个单线程化的线程池，使用一个唯一的工作线程执行任务，保证所有任务按照指定顺序（先入先出或者优先级）执行
- Executors.newSingleThreadScheduledExecutor:创建一个单线程化的线程池，支持定时、周期性的任务执行
- Executors.newWorkStealingPool：创建一个具有并行级别的work-stealing线程池

3.线程池实例的几种状态

- Running:运行状态，能接收新提交的任务，并且也能处理阻塞队列中的任务
- Shutdown: 关闭状态，不能再接收新提交的任务，但是可以处理阻塞队列中已经保存的任务，当线程池处于Running状态时，调用shutdown()方法会使线程池进入该状态
- Stop: 不能接收新任务，也不能处理阻塞队列中已经保存的任务，会中断正在处理任务的线程，如果线程池处于Running或Shutdown状态，调用shutdownNow()方法，会使线程池进入该状态
- Tidying: 如果所有的任务都已经终止，有效线程数为0（阻塞队列为空，线程池中的工作线程数量为0），线程池就会进入该状态。
- Terminated: 处于Tidying状态的线程池调用terminated()方法，会使用线程池进入该状态

注意：不需要对线程池的状态做特殊的处理，线程池的状态是线程池内部根据方法自行定义和处理的。

4.合理配置线程的一些建议

（1）CPU密集型任务，就需要尽量压榨CPU，参考值可以设置为NCPU+1(CPU的数量加1)。
（2）IO密集型任务，参考值可以设置为2*NCPU（CPU数量乘以2）

线程池最核心的类之一——ThreadPoolExecutor

1.构造方法

ThreadPoolExecutor参数最多的构造方法如下：

```
public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory,
                              RejectedExecutionHandler rejectHandler)
```

其他的构造方法都是调用的这个构造方法来实例化对象，可以说，我们直接分析这个方法之后，其他的构造方法我们也明白是怎么回事了！接下来，就对此构造方法进行详细的分析。

注意：为了更加深入的分析ThreadPoolExecutor类的构造方法，会适当调整参数的顺序进行解析，以便于大家更能深入的理解ThreadPoolExecutor构造方法中每个参数的作用。

上述构造方法接收如下参数进行初始化：

（1）corePoolSize：核心线程数量。

（2）maximumPoolSize：最大线程数。

（3）workQueue：阻塞队列，存储等待执行的任务，很重要，会对线程池运行过程产生重大影响。

其中，上述三个参数的关系如下所示：

- 如果运行的线程数小于corePoolSize，直接创建新线程处理任务，即使线程池中的其他线程是空闲的。
- 如果运行的线程数大于等于corePoolSize，并且小于maximumPoolSize，此时，只有当workQueue满时，才会创建新的线程处理任务。
- 如果设置的corePoolSize与maximumPoolSize相同，那么创建的线程池大小是固定的，此时，如果有新任务提交，并且workQueue没有满时，就把请求放入到workQueue中，等待空闲的线程，从workQueue中取出任务进行处理。
- 如果运行的线程数量大于maximumPoolSize，同时，workQueue已经满了，会通过拒绝策略参数rejectHandler来指定处理策略。

根据上述三个参数的配置，线程池会对任务进行如下处理方式：

当提交一个新的任务到线程池时，线程池会根据当前线程池中正在运行的线程数量来决定该任务的处理方式。处理方式总共有三种：直接切换、使用无限队列、使用有界队列。

- 直接切换常用的队列就是SynchronousQueue。
- 使用无限队列就是使用基于链表的队列，比如：LinkedBlockingQueue，如果使用这种方式，线程池中创建的最大线程数就是corePoolSize，此时maximumPoolSize不会起作用。当线程池中所有的核心线程都是运行状态时，提交新任务，就会放入等待队列中。
- 使用有界队列使用的是ArrayBlockingQueue，使用这种方式可以将线程池的最大线程数量限制为maximumPoolSize，可以降低资源的消耗。但是，这种方式使得线程池对线程的调度更困难，因为线程池和队列的容量都是有限的了。

根据上面三个参数，我们可以简单得出如何降低系统资源消耗的一些措施：

- 如果想降低系统资源的消耗，包括CPU使用率，操作系统资源的消耗，上下文环境切换的开销等，可以设置一个较大的队列容量和较小的线程池容量。这样，会降低线程处理任务的吞吐量。
- 如果提交的任务经常发生阻塞，可以考虑调用设置最大线程数的方法，重新设置线程池最大线程数。如果队列的容量设置的较小，通常需要将线程池的容量设置的大一些，这样，CPU的使用率会高些。如果线程池的容量设置的过大，并发量就会增加，则需要考虑线程调度的问题，反而可能会降低处理任务的吞吐量。

接下来，我们继续看ThreadPoolExecutor的构造方法的参数。

（4）keepAliveTime：线程没有任务执行时最多保持多久时间终止
当线程池中的线程数量大于corePoolSize时，如果此时没有新的任务提交，核心线程外的线程不会立即销毁，需要等待，直到等待的时间超过了keepAliveTime就会终止。

（5）unit：keepAliveTime的时间单位

（6）threadFactory：线程工厂，用来创建线程
默认会提供一个默认的工厂来创建线程，当使用默认的工厂来创建线程时，会使新创建的线程具有相同的优先级，并且是非守护的线程，同时也设置了线程的名称

（7）rejectHandler：拒绝处理任务时的策略

如果workQueue阻塞队列满了，并且没有空闲的线程池，此时，继续提交任务，需要采取一种策略来处理这个任务。

线程池总共提供了四种策略：

- 直接抛出异常，这也是默认的策略。实现类为AbortPolicy。
- 用调用者所在的线程来执行任务。实现类为CallerRunsPolicy。
- 丢弃队列中最靠前的任务并执行当前任务。实现类为DiscardOldestPolicy。
- 直接丢弃当前任务。实现类为DiscardPolicy。

2.ThreadPoolExecutor提供的启动和停止任务的方法

（1）execute():提交任务，交给线程池执行
（2）submit():提交任务，能够返回执行结果 execute+Future
（3）shutdown():关闭线程池，等待任务都执行完
（4）shutdownNow():立即关闭线程池，不等待任务执行完

3.ThreadPoolExecutor提供的适用于监控的方法

（1）getTaskCount()：线程池已执行和未执行的任务总数
（2）getCompletedTaskCount()：已完成的任务数量
（3）getPoolSize()：线程池当前的线程数量
（4）getCorePoolSize()：线程池核心线程数
（5）getActiveCount():当前线程池中正在执行任务的线程数量

深度解析线程池中重要的顶层接口和抽象类

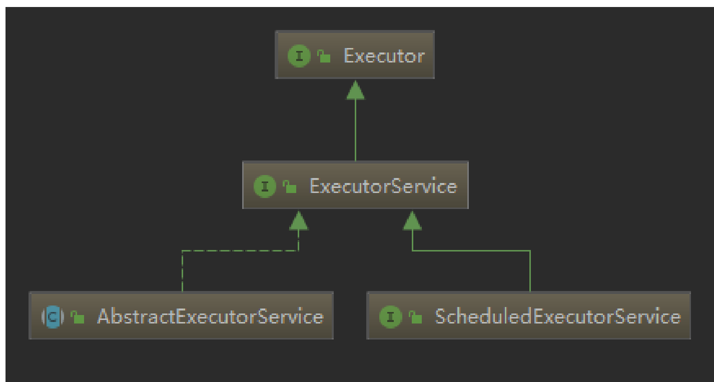
前面我们从整体上介绍了Java的线程池。如果细细品味线程池的底层源码实现，你会发现整个线程池体系的设计是非常优雅的！这些代码的设计值得我们去细细品味和研究，从中学习优雅代码的设计规范，形成自己的设计思想，为我所用！哈哈，说多了，接下来，我们就来看看线程池中那些非常重要的接口和抽象类，深度分析下线程池中是如何将抽象这一思想运用的淋漓尽致的！

通过对线程池中接口和抽象类的分析，你会发现，整个线程池设计的是如此的优雅和强大，从线程池的代码设计中，我们学到的不只是代码而已！！

题外话：膜拜Java大神Doug Lea，Java中的并发包正是这位老爷子写的，他是这个世界上对Java影响力最大的一个人。

接口和抽象类总览

说起线程池中提供的重要的接口和抽象类，基本上就是如下图所示的接口和类。



接口与类的简单说明：

- Executor接口：这个接口也是整个线程池中最顶层的接口，提供了一个无返回值的提交任务的方法。
- ExecutorService接口：派生自Executor接口，扩展了很过功能，例如关闭线程池，提交任务并返回结果数据、唤醒线程池中的任务等。
- AbstractExecutorService抽象类：派生自ExecutorService接口，实现了几个非常实现的方法，供子类进行调用。
- ScheduledExecutorService定时任务接口，派生自ExecutorService接口，拥有ExecutorService接口定义的全部方法，并扩展了定时任务相关的方法。

接下来，我们就分别从源码角度来看下这些接口和抽象类从顶层设计上提供了哪些功能。

Executor接口

Executor接口的源码如下所示。

```
public interface Executor {
    //提交运行任务，参数为Runnable接口对象，无返回值
    void execute(Runnable command);
}
```

从源码可以看出，Executor接口非常简单，只提供了一个无返回值的提交任务的execute(Runnable)方法。

由于这个接口过于简单，我们无法得知线程池的执行结果数据，如果我们不再使用线程池，也无法通过Executor接口来关闭线程池。此时，我们就需要ExecutorService接口的支持了。

ExecutorService接口

ExecutorService接口是非定时任务类线程池的核心接口，通过ExecutorService接口能够向线程池中提交任务（支持有返回结果和无返回结果两种方式）、关闭线程池、唤醒线程池中的任务等。ExecutorService接口的源码如下所示。

```
package java.util.concurrent;
import java.util.List;
import java.util.Collection;
public interface ExecutorService extends Executor {

    //关闭线程池，线程池中不再接受新提交的任务，但是之前提交的任务继续运行，直到完成
    void shutdown();

    //关闭线程池，线程池中不再接受新提交的任务，会尝试停止线程池中正在执行的任务。
```

```java
    List<Runnable> shutdownNow();

    //判断线程池是否已经关闭
    boolean isShutdown();

    //判断线程池中的所有任务是否结束，只有在调用shutdown或者shutdownNow方法之后调用此方法才会返回true。
    boolean isTerminated();

    //等待线程池中的所有任务执行结束，并设置超时时间
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;

    //提交一个Callable接口类型的任务，返回一个Future类型的结果
    <T> Future<T> submit(Callable<T> task);

    //提交一个Callable接口类型的任务，并且给定一个泛型类型的接收结果数据参数，返回一个Future类型的结果
    <T> Future<T> submit(Runnable task, T result);

    //提交一个Runnable接口类型的任务，返回一个Future类型的结果
    Future<?> submit(Runnable task);

    //批量提交任务并获得他们的future，Task列表与Future列表一一对应
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

    //批量提交任务并获得他们的future，并限定处理所有任务的时间
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                  long timeout, TimeUnit unit) throws InterruptedException;

    //批量提交任务并获得一个已经成功执行的任务的结果
    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

    //批量提交任务并获得一个已经成功执行的任务的结果，并限定处理任务的时间
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

关于ExecutorService接口中每个方法的含义，直接上述接口源码中的注释即可，这些接口方法都比较简单，我就不一一重复列举描述了。这个接口也是我们在使用非定时任务类的线程池中最常使用的接口。

AbstractExecutorService抽象类

AbstractExecutorService类是一个抽象类，派生自ExecutorService接口，在其基础上实现了几个比较实用的方法，提供给子类进行调用。我们还是来看下AbstractExecutorService类的源码。

注意：大家可以到java.util.concurrent包下查看完整的AbstractExecutorService类的源码，这里，我将AbstractExecutorService源码进行拆解，详解每个方法的作用。

- newTaskFor方法

```java
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}

protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    return new FutureTask<T>(callable);
}
```

RunnableFuture类用于获取执行结果，在实际使用时，我们经常使用的是它的子类FutureTask，newTaskFor方法的作用就是将任务封装成FutureTask对象，后续将FutureTask对象提交到线程池。

- doInvokeAny方法

```java
private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,
                          boolean timed, long nanos)
    throws InterruptedException, ExecutionException, TimeoutException {
    //提交的任务为空，抛出空指针异常
    if (tasks == null)
```

```
        throw new NullPointerException();
    //记录待执行的任务的剩余数量
    int ntasks = tasks.size();
    //任务集合中的数据为空，抛出非法参数异常
    if (ntasks == 0)
        throw new IllegalArgumentException();
    ArrayList<Future<T>> futures = new ArrayList<Future<T>>(ntasks);
    //以当前实例对象作为参数构建ExecutorCompletionService对象
    // ExecutorCompletionService负责执行任务，后面调用用poll返回第一个执行结果
    ExecutorCompletionService<T> ecs =
        new ExecutorCompletionService<T>(this);

    try {
        // 记录可能抛出的执行异常
        ExecutionException ee = null;
        // 初始化超时时间
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        Iterator<? extends Callable<T>> it = tasks.iterator();

        //提交任务，并将返回的结果数据添加到futures集合中
        //提交一个任务主要是确保在进入循环之前开始一个任务
        futures.add(ecs.submit(it.next()));
        --ntasks;
        //记录正在执行的任务数量
        int active = 1;

        for (;;) {
            //从完成任务的BlockingQueue队列中获取并移除下一个将要完成的任务的结果。
            //如果BlockingQueue队列中中的数据为空，则返回null
            //这里的poll()方法是非阻塞方法
            Future<T> f = ecs.poll();
            //获取的结果为空
            if (f == null) {
                //集合中仍有未执行的任务数量
                if (ntasks > 0) {
                    //未执行的任务数量减1
                    --ntasks;
                    //提交完成并将结果添加到futures集合中
                    futures.add(ecs.submit(it.next()));
                    //正在执行的任务数量加•1
                    ++active;
                }
                //所有任务执行完成，并且返回了结果数据，则退出循环
                //之所以处理active为0的情况，是因为poll()方法是非阻塞方法，可能导致未返回结果时active为0
                else if (active == 0)
                    break;
                //如果timed为true，则执行获取结果数据时设置超时时间，也就是超时获取结果表示
                else if (timed) {
                    f = ecs.poll(nanos, TimeUnit.NANOSECONDS);
                    if (f == null)
                        throw new TimeoutException();
                    nanos = deadline - System.nanoTime();
                }
                //没有设置超时，并且所有任务都被提交了，则一直阻塞，直到返回一个执行结果
                else
                    f = ecs.take();
            }
            //获取到执行结果，则将正在执行的任务减1，从Future中获取结果并返回
            if (f != null) {
                --active;
                try {
                    return f.get();
                } catch (ExecutionException eex) {
                    ee = eex;
                } catch (RuntimeException rex) {
                    ee = new ExecutionException(rex);
                }
            }
        }

        if (ee == null)
```

```java
            ee = new ExecutionException();
        throw ee;

    } finally {
        //如果从所有执行的任务中获取到一个结果数据，则取消所有执行的任务，不再向下执行
        for (int i = 0, size = futures.size(); i < size; i++)
            futures.get(i).cancel(true);
    }
}
```

这个方法是批量执行线程池的任务，最终返回一个结果数据的核心方法，通过源代码的分析，我们可以发现，这个方法只要获取到一个结果数据，就会取消线程池中所有运行的任务，并将结果数据返回。这就好比是很多要进入一个居民小区一样，只要有一个人有门禁卡，门卫就不再检查其他人是否有门禁卡，直接放行。

在上述代码中，我们看到提交任务使用的ExecutorCompletionService对象的submit方法，我们再来看下ExecutorCompletionService类中的submit方法，如下所示。

```java
public Future<V> submit(Callable<V> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task);
    executor.execute(new QueueingFuture(f));
    return f;
}

public Future<V> submit(Runnable task, V result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task, result);
    executor.execute(new QueueingFuture(f));
    return f;
}
```

可以看到，ExecutorCompletionService类中的submit方法本质上调用的还是Executor接口的execute方法。

- invokeAny方法

```java
public <T> T invokeAny(Collection<? extends Callable<T>> tasks)
    throws InterruptedException, ExecutionException {
    try {
        return doInvokeAny(tasks, false, 0);
    } catch (TimeoutException cannotHappen) {
        assert false;
        return null;
    }
}

public <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                       long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException {
    return doInvokeAny(tasks, true, unit.toNanos(timeout));
}
```

这两个invokeAny方法本质上都是在调用doInvokeAny方法，在线程池中提交多个任务，只要返回一个结果数据即可。

直接看上面的代码，大家可能有点晕。这里，我举一个例子，我们在使用线程池的时候，可能会启动多个线程去执行各自的任务，比如线程A负责task_a，线程B负责task_b，这样可以大规模提升系统处理任务的速度。如果我们希望其中一个线程执行完成返回结果数据时立即返回，而不需要再让其他线程继续执行任务。此时，就可以使用invokeAny方法。

- invokeAll方法

```java
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    throws InterruptedException {
    if (tasks == null)
        throw new NullPointerException();
    ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
    //标识所有任务是否完成
    boolean done = false;
    try {
        //遍历所有任务
        for (Callable<T> t : tasks) {
            将每个任务封装成RunnableFuture对象提交任务
```

```
            RunnableFuture<T> f = newTaskFor(t);
            //将结果数据添加到futures集合中
            futures.add(f);
            //执行任务
            execute(f);
        }
        //遍历结果数据集合
        for (int i = 0, size = futures.size(); i < size; i++) {
            Future<T> f = futures.get(i);
            //任务没有完成
            if (!f.isDone()) {
                try {
                    //阻塞等待任务完成并返回结果
                    f.get();
                } catch (CancellationException ignore) {
                } catch (ExecutionException ignore) {
                }
            }
        }
        //任务完成（不管是正常结束还是异常完成）
        done = true;
        //返回结果数据集合
        return futures;
    } finally {
        //如果发生中断异常InterruptedException 则取消已经提交的任务
        if (!done)
            for (int i = 0, size = futures.size(); i < size; i++)
                futures.get(i).cancel(true);
    }
}

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                     long timeout, TimeUnit unit)
    throws InterruptedException {
    if (tasks == null)
        throw new NullPointerException();
    long nanos = unit.toNanos(timeout);
    ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
    boolean done = false;
    try {
        for (Callable<T> t : tasks)
            futures.add(newTaskFor(t));

        final long deadline = System.nanoTime() + nanos;
        final int size = futures.size();

        for (int i = 0; i < size; i++) {
            execute((Runnable)futures.get(i));
            // 在添加执行任务时超时判断，如果超时则立刻返回futures集合
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L)
                return futures;
        }
         // 遍历所有任务
        for (int i = 0; i < size; i++) {
            Future<T> f = futures.get(i);
            if (!f.isDone()) {
                //对结果进行判断时进行超时判断
                if (nanos <= 0L)
                    return futures;
                try {
                    f.get(nanos, TimeUnit.NANOSECONDS);
                } catch (CancellationException ignore) {
                } catch (ExecutionException ignore) {
                } catch (TimeoutException toe) {
                    return futures;
                }
                //重置任务的超时时间
                nanos = deadline - System.nanoTime();
            }
        }
```

```
            done = true;
            return futures;
        } finally {
            if (!done)
                for (int i = 0, size = futures.size(); i < size; i++)
                    futures.get(i).cancel(true);
        }
    }
```

invokeAll方法同样实现了无超时时间设置和有超时时间设置的逻辑。

无超时时间设置的invokeAll方法总体逻辑为：将所有任务封装成RunnableFuture对象，调用execute方法执行任务，将返回的结果数据添加到futures集合，之后对futures集合进行遍历判断，检测任务是否完成，如果没有完成，则调用get方法阻塞任务，直到返回结果数据，此时会忽略异常。最终在finally代码块中对所有任务是否完成的标识进行判断，如果存在未完成的任务，则取消已经提交的任务。

有超时设置的invokeAll方法总体逻辑与无超时时间设置的invokeAll方法总体逻辑基本相同，只是在两个地方添加了超时的逻辑判断。一个是在添加执行任务时进行超时判断，如果超时，则立刻返回futures集合；另一个是每次对结果数据进行判断时添加了超时处理逻辑。

invokeAll方法中本质上还是调用Executor接口的execute方法来提交任务。

- submit方法

submit方法的逻辑比较简单，就是将任务封装成RunnableFuture对象并提交，执行任务后返回Future结果数据。如下所示。

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}

public <T> Future<T> submit(Runnable task, T result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task, result);
    execute(ftask);
    return ftask;
}

public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

从源码中可以看出submit方法提交任务时，本质上还是调用的Executor接口的execute方法。

综上所述，在非定时任务类的线程池中提交任务时，本质上都是调用的Executor接口的execute方法。至于调用的是哪个具体实现类的execute方法，我们在后面的文章中深入分析。

ScheduledExecutorService接口

ScheduledExecutorService接口派生自ExecutorService接口，继承了ExecutorService接口的所有功能，并提供了定时处理任务的能力，ScheduledExecutorService接口的源代码比较简单，如下所示。

```
package java.util.concurrent;

public interface ScheduledExecutorService extends ExecutorService {

    //延时delay时间来执行command任务，只执行一次
    public ScheduledFuture<?> schedule(Runnable command,
                                       long delay, TimeUnit unit);

    //延时delay时间来执行callable任务，只执行一次
    public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                                           long delay, TimeUnit unit);

    //延时initialDelay时间首次执行command任务，之后每隔period时间执行一次
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
```

```java
                                                    long initialDelay,
                                                    long period,
                                                    TimeUnit unit);

    //延时initialDelay时间首次执行command任务，之后每延时delay时间执行一次
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                       long initialDelay,
                                                       long delay,
                                                       TimeUnit unit);

}
```

至此，我们分析了线程池体系中重要的顶层接口和抽象类。

通过对这些顶层接口和抽象类的分析，我们需要从中感悟并体会软件开发中的抽象思维，深入理解抽象思维在具体编码中的实现，最终，形成自己的编程思维，运用到实际的项目中，这也是我们能够从源码中所能学到的众多细节之一。这也是高级或资深工程师和架构师必须了解源码细节的原因之一。

从源码角度分析创建线程池究竟有哪些方式

前言

在Java的高并发领域，线程池一直是一个绕不开的话题。有些童鞋一直在使用线程池，但是，对于如何创建线程池仅仅停留在使用Executors工具类的方式，那么，创建线程池究竟存在哪几种方式呢？就让我们一起从创建线程池的源码来深入分析究竟有哪些方式可以创建线程池。

使用Executors工具类创建线程池

在创建线程池时，初学者用的最多的就是Executors 这个工具类，而使用这个工具类创建线程池时非常简单的，不需要关注太多的线程池细节，只需要传入必要的参数即可。Executors 工具类提供了几种创建线程池的方法，如下所示。

- Executors.newCachedThreadPool：创建一个可缓存的线程池，如果线程池的大小超过了需要，可以灵活回收空闲线程，如果没有可回收线程，则新建线程
- Executors.newFixedThreadPool：创建一个定长的线程池，可以控制线程的最大并发数，超出的线程会在队列中等待
- Executors.newScheduledThreadPool：创建一个定长的线程池，支持定时、周期性的任务执行
- Executors.newSingleThreadExecutor: 创建一个单线程化的线程池，使用一个唯一的工作线程执行任务，保证所有任务按照指定顺序（先入先出或者优先级）执行
- Executors.newSingleThreadScheduledExecutor:创建一个单线程化的线程池，支持定时、周期性的任务执行
- Executors.newWorkStealingPool：创建一个具有并行级别的work-stealing线程池

其中，Executors.newWorkStealingPool方法是Java 8中新增的创建线程池的方法，它能够为线程池设置并行级别，具有更高的并发度和性能。除了此方法外，其他创建线程池的方法本质上调用的是ThreadPoolExecutor类的构造方法。

例如，我们可以使用如下代码创建线程池。

```java
Executors.newWorkStealingPool();
Executors.newCachedThreadPool();
Executors.newScheduledThreadPool(3);
```

使用ThreadPoolExecutor类创建线程池

从代码结构上看ThreadPoolExecutor类继承自AbstractExecutorService，也就是说，ThreadPoolExecutor类具有AbstractExecutorService类的全部功能。

既然Executors工具类中创建线程池大部分调用的都是ThreadPoolExecutor类的构造方法，所以，我们也可以直接调用ThreadPoolExecutor类的构造方法来创建线程池，而不再使用Executors工具类。接下来，我们一起看下ThreadPoolExecutor类的构造方法。

ThreadPoolExecutor类中的所有构造方法如下所示。

```java
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                         BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
         Executors.defaultThreadFactory(), defaultHandler);
```

```
}

public ThreadPoolExecutor(int corePoolSize,
                int maximumPoolSize,
                long keepAliveTime,
                TimeUnit unit,
                BlockingQueue<Runnable> workQueue,
                    ThreadFactory threadFactory) {
this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
     threadFactory, defaultHandler);
}

public ThreadPoolExecutor(int corePoolSize,
                int maximumPoolSize,
                long keepAliveTime,
                    TimeUnit unit,
                BlockingQueue<Runnable> workQueue,
                RejectedExecutionHandler handler) {
this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
     Executors.defaultThreadFactory(), handler);
}

public ThreadPoolExecutor(int corePoolSize,
                int maximumPoolSize,
                long keepAliveTime,
                TimeUnit unit,
                    BlockingQueue<Runnable> workQueue,
                ThreadFactory threadFactory,
                RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

由ThreadPoolExecutor类的构造方法的源代码可知，创建线程池最终调用的构造方法如下。

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
              long keepAliveTime, TimeUnit unit,
              BlockingQueue<Runnable> workQueue,
              ThreadFactory threadFactory,
                  RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
```

```
}
```

大家可以自行调用ThreadPoolExecutor类的构造方法来创建线程池。例如，我们可以使用如下形式创建线程池。

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                       60L, TimeUnit.SECONDS,
                       new SynchronousQueue<Runnable>());
```

使用ForkJoinPool类创建线程池

在Java8的Executors工具类中，新增了如下创建线程池的方式。

```
public static ExecutorService newWorkStealingPool(int parallelism) {
    return new ForkJoinPool
        (parallelism,
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
}

public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool
        (Runtime.getRuntime().availableProcessors(),
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
}
```

从源代码可以可以，本质上调用的是ForkJoinPool类的构造方法类创建线程池，而从代码结构上来看ForkJoinPool类继承自AbstractExecutorService抽象类。接下来，我们看下ForkJoinPool类的构造方法。

```
public ForkJoinPool() {
    this(Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors()),
         defaultForkJoinWorkerThreadFactory, null, false);
}
 public ForkJoinPool(int parallelism) {
    this(parallelism, defaultForkJoinWorkerThreadFactory, null, false);
}

public ForkJoinPool(int parallelism,
                ForkJoinWorkerThreadFactory factory,
                UncaughtExceptionHandler handler,
                boolean asyncMode) {
    this(checkParallelism(parallelism),
         checkFactory(factory),
         handler,
         asyncMode ? FIFO_QUEUE : LIFO_QUEUE,
         "ForkJoinPool-" + nextPoolId() + "-worker-");
    checkPermission();
}

private ForkJoinPool(int parallelism,
                 ForkJoinWorkerThreadFactory factory,
                 UncaughtExceptionHandler handler,
                 int mode,
                 String workerNamePrefix) {
    this.workerNamePrefix = workerNamePrefix;
    this.factory = factory;
    this.ueh = handler;
    this.config = (parallelism & SMASK) | mode;
    long np = (long)(-parallelism); // offset ctl counts
    this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_MASK);
}
```

通过查看源代码得知，ForkJoinPool的构造方法，最终调用的是如下私有构造方法。

```java
private ForkJoinPool(int parallelism,
                     ForkJoinWorkerThreadFactory factory,
                     UncaughtExceptionHandler handler,
                     int mode,
                     String workerNamePrefix) {
    this.workerNamePrefix = workerNamePrefix;
    this.factory = factory;
    this.ueh = handler;
    this.config = (parallelism & SMASK) | mode;
    long np = (long)(-parallelism); // offset ctl counts
    this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_MASK);
}
```

其中，各参数的含义如下所示。

- parallelism：并发级别。
- factory：创建线程的工厂类对象。
- handler：当线程池中的线程抛出未捕获的异常时，统一使用UncaughtExceptionHandler对象处理。
- mode：取值为FIFO_QUEUE或者LIFO_QUEUE。
- workerNamePrefix：执行任务的线程名称的前缀。

当然，私有构造方法虽然是参数最多的一个方法，但是其不会直接对外方法，我们可以使用如下方式创建线程池。

```java
new ForkJoinPool();
new ForkJoinPool(Runtime.getRuntime().availableProcessors());
new ForkJoinPool(Runtime.getRuntime().availableProcessors(),
             ForkJoinPool.defaultForkJoinWorkerThreadFactory,
             null, true);
```

使用ScheduledThreadPoolExecutor类创建线程池

在Executors工具类中存在如下方法类创建线程池。

```java
public static ScheduledExecutorService newSingleThreadScheduledExecutor() {
    return new DelegatedScheduledExecutorService
        (new ScheduledThreadPoolExecutor(1));
}

public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory) {
    return new DelegatedScheduledExecutorService
        (new ScheduledThreadPoolExecutor(1, threadFactory));
}

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(
        int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
```

从源码来看，这几个方法本质上调用的都是ScheduledThreadPoolExecutor类的构造方法，ScheduledThreadPoolExecutor中存在的构造方法如下所示。

```java
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue());
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), threadFactory);
}

public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), handler);
```

```
}

public ScheduledThreadPoolExecutor(int corePoolSize,ThreadFactory threadFactory,
RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), threadFactory, handler);
}
```

ThreadPoolExecutor类继承自ThreadPoolExecutor类，本质上还是调用ThreadPoolExecutor类的构造方法，只不过此时传递的队列为DelayedWorkQueue。我们可以直接调用ScheduledThreadPoolExecutor类的构造方法来创建线程池，例如以如下形式创建线程池。

```
new ScheduledThreadPoolExecutor(3)
```

通过源码深度解析ThreadPoolExecutor类是如何保证线程池正确运行的

问题：

对于线程池的核心类ThreadPoolExecutor来说，有哪些重要的属性和内部类为线程池的正确运行提供重要的保障呢？

ThreadPoolExecutor类中的重要属性

在ThreadPoolExecutor类中，存在几个非常重要的属性和方法，接下来，我们就介绍下这些重要的属性和方法。

ctl相关的属性

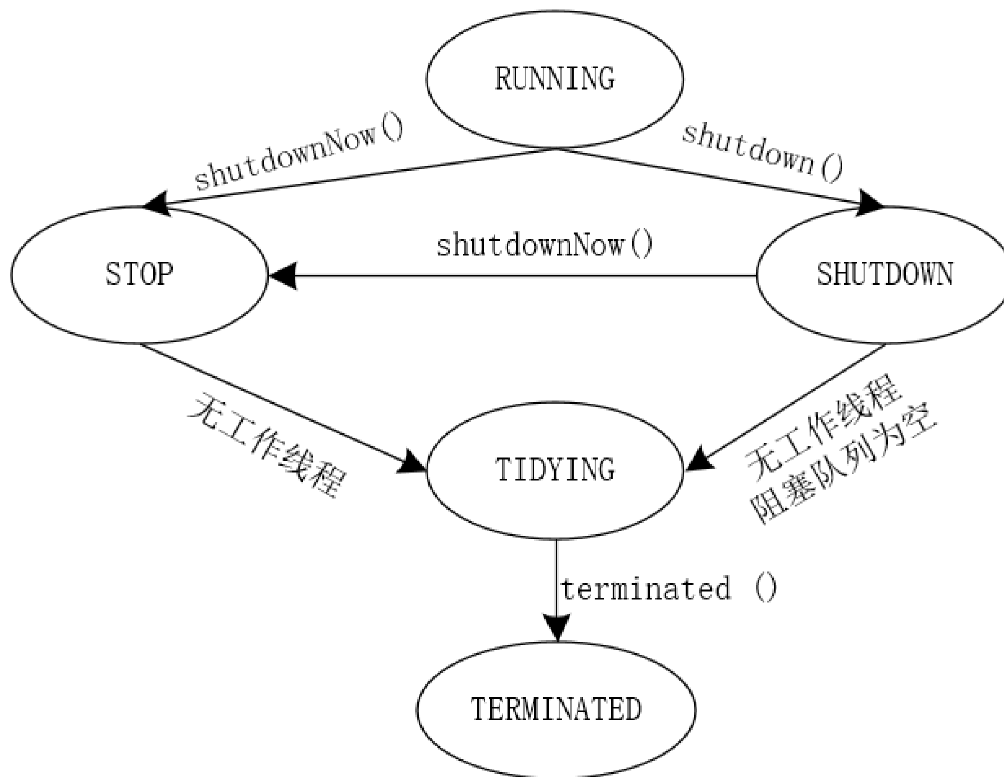
AtomicInteger类型的常量ctl是贯穿线程池整个生命周期的重要属性，它是一个原子类对象，主要用来保存线程的数量和线程池的状态，我们看下与这个属性相关的代码如下所示。

```
//主要用来保存线程数量和线程池的状态，高3位保存线程状态，低29位保存线程数量
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//线程池中线程的数量的位数（32-3）
private static final int COUNT_BITS = Integer.SIZE - 3;
//表示线程池中的最大线程数量
//将数字1的二进制值向右移29位，再减去1
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
//线程池的运行状态
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;
//获取线程状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }
//获取线程数量
private static int workerCountOf(int c)  { return c & CAPACITY; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
private static boolean runStateLessThan(int c, int s) {
    return c < s;
}
private static boolean runStateAtLeast(int c, int s) {
    return c >= s;
}
private static boolean isRunning(int c) {
    return c < SHUTDOWN;
}
private boolean compareAndIncrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect + 1);
}
private boolean compareAndDecrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect - 1);
}
private void decrementWorkerCount() {
    do {} while (! compareAndDecrementWorkerCount(ctl.get()));
}
```

对于线程池的各状态说明如下所示。

- RUNNING:运行状态，能接收新提交的任务，并且也能处理阻塞队列中的任务
- SHUTDOWN: 关闭状态，不能再接收新提交的任务，但是可以处理阻塞队列中已经保存的任务，当线程池处于RUNNING状态时，调用shutdown()方法会使线程池进入该状态
- STOP: 不能接收新任务，也不能处理阻塞队列中已经保存的任务，会中断正在处理任务的线程，如果线程池处于RUNNING或SHUTDOWN状态，调用shutdownNow()方法，会使线程池进入该状态
- TIDYING: 如果所有的任务都已经终止，有效线程数为0（阻塞队列为空，线程池中的工作线程数量为0），线程池就会进入该状态。
- TERMINATED: 处于TIDYING状态的线程池调用terminated ()方法，会使用线程池进入该状态

也可以按照ThreadPoolExecutor类的注释，将线程池的各状态之间的转化总结成如下图所示。



- RUNNING -> SHUTDOWN：显式调用shutdown()方法, 或者隐式调用了finalize()方法
- (RUNNING or SHUTDOWN) -> STOP：显式调用shutdownNow()方法
- SHUTDOWN -> TIDYING：当线程池和任务队列都为空的时候
- STOP -> TIDYING：当线程池为空的时候
- TIDYING -> TERMINATED：当 terminated() hook 方法执行完成时候

其他重要属性

除了ctl相关的属性外，ThreadPoolExecutor类中其他一些重要的属性如下所示。

```
//用于存放任务的阻塞队列
private final BlockingQueue<Runnable> workQueue;
//可重入锁
private final ReentrantLock mainLock = new ReentrantLock();
//存放线程池中线程的集合，访问这个集合时，必须获得mainLock锁
private final HashSet<Worker> workers = new HashSet<Worker>();
//在锁内部阻塞等待条件完成
private final Condition termination = mainLock.newCondition();
//线程工厂，以此来创建新线程
private volatile ThreadFactory threadFactory;
//拒绝策略
private volatile RejectedExecutionHandler handler;
//默认的拒绝策略
private static final RejectedExecutionHandler defaultHandler = new AbortPolicy();
```

ThreadPoolExecutor类中的重要内部类

在ThreadPoolExecutor类中存在对于线程池的执行至关重要的内部类，Worker内部类和拒绝策略内部类。接下来，我们分别看这些内部类。

Worker内部类

Worker类从源代码上来看，实现了Runnable接口，说明其本质上是一个用来执行任务的线程，接下来，我们看下Worker类的源代码，如下所示。

```
private final class Worker extends AbstractQueuedSynchronizer implements Runnable{
    private static final long serialVersionUID = 6138294804551838833L;
    //真正执行任务的线程
    final Thread thread;
    //第一个Runnable任务，如果在创建线程时指定了需要执行的第一个任务
    //则第一个任务会存放在此变量中，此变量也可以为null
    //如果为null，则线程启动后，通过getTask方法到BlockingQueue队列中获取任务
    Runnable firstTask;
    //用于存放此线程完全的任务数，注意：使用了volatile关键字
    volatile long completedTasks;

    //Worker类唯一的构造放大，传递的firstTask可以为null
    Worker(Runnable firstTask) {
        //防止在调用runWorker之前被中断
        setState(-1);
        this.firstTask = firstTask;
        //使用ThreadFactory 来创建一个新的执行任务的线程
        this.thread = getThreadFactory().newThread(this);
    }
    //调用外部ThreadPoolExecutor类的runWorker方法执行任务
    public void run() {
        runWorker(this);
    }

    //是否获取到锁
    //state=0表示锁未被获取
    //state=1表示锁被获取
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }

    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }

    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }

    void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}
```

在Worker类的构造方法中，可以看出，首先将同步状态state设置为-1，设置为-1是为了防止runWorker方法运行之前被中断。这是因为如果其他线程调用线程池的shutdownNow()方法时，如果Worker类中的state状态的值大于0，则会中断线程，如果state状态的值为-1，则不会中断线程。

Worker类实现了Runnable接口，需要重写run方法，而Worker的run方法本质上调用的是ThreadPoolExecutor类的runWorker方法，在runWorker方法中，会首先调用unlock方法，该方法会将state置为0，所以这个时候调用shutDownNow方法就会中断当前线程，而这个时候已经进入了runWork方法，就不会在还没有执行runWorker方法的时候就中断线程。

注意：大家需要重点理解Worker类的实现

拒绝策略内部类

在线程池中，如果workQueue阻塞队列满了，并且没有空闲的线程池，此时，继续提交任务，需要采取一种策略来处理这个任务。而线程池总共提供了四种策略，如下所示。

- 直接抛出异常，这也是默认的策略。实现类为AbortPolicy。
- 用调用者所在的线程来执行任务。实现类为CallerRunsPolicy。
- 丢弃队列中最靠前的任务并执行当前任务。实现类为DiscardOldestPolicy。
- 直接丢弃当前任务。实现类为DiscardPolicy。

在ThreadPoolExecutor类中提供了4个内部类来默认实现对应的策略，如下所示。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {

    public CallerRunsPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}

public static class AbortPolicy implements RejectedExecutionHandler {

    public AbortPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        throw new RejectedExecutionException("Task " + r.toString() + " rejected from " +
e.toString());
    }
}

public static class DiscardPolicy implements RejectedExecutionHandler {

    public DiscardPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    }
}

public static class DiscardOldestPolicy implements RejectedExecutionHandler {

    public DiscardOldestPolicy() { }


    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            e.getQueue().poll();
            e.execute(r);
        }
    }
}
```

我们也可以通过实现RejectedExecutionHandler接口，并重写RejectedExecutionHandler接口的rejectedExecution方法来自定义拒绝策略，在创建线程池时，调用ThreadPoolExecutor的构造方法，传入我们自己写的拒绝策略。

例如，自定义的拒绝策略如下所示。

```java
public class CustomPolicy implements RejectedExecutionHandler {

    public CustomPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            System.out.println("使用调用者所在的线程来执行任务")
            r.run();
        }
    }
}
```

使用自定义拒绝策略创建线程池。

```java
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                       60L, TimeUnit.SECONDS,
                       new SynchronousQueue<Runnable>(),
                       Executors.defaultThreadFactory(),
               new CustomPolicy());
```

通过ThreadPoolExecutor类的源码深度解析线程池执行任务的核心流程

核心逻辑概述

ThreadPoolExecutor是Java线程池中最核心的类之一，它能够保证线程池按照正常的业务逻辑执行任务，并通过原子方式更新线程池每个阶段的状态。

ThreadPoolExecutor类中存在一个workers工作线程集合，用户可以向线程池中添加需要执行的任务，workers集合中的工作线程可以直接执行任务，或者从任务队列中获取任务后执行。ThreadPoolExecutor类中提供了整个线程池从创建到执行任务，再到消亡的整个流程方法。本文，就结合ThreadPoolExecutor类的源码深度分析线程池执行任务的整体流程。

在ThreadPoolExecutor类中，线程池的逻辑主要体现在execute(Runnable)方法，addWorker(Runnable, boolean)方法，addWorkerFailed(Worker)方法和拒绝策略上，接下来，我们就深入分析这几个核心方法。

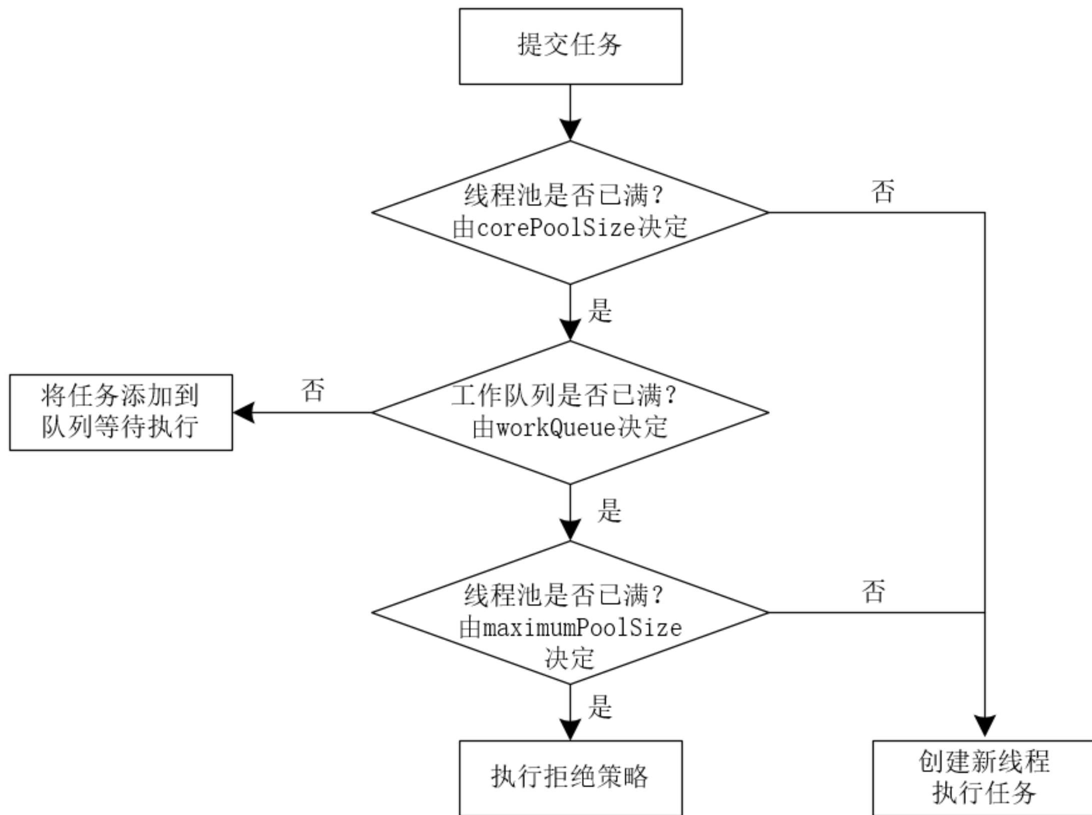
execute(Runnable)方法

execute(Runnable)方法的作用是提交Runnable类型的任务到线程池中。我们先看下execute(Runnable)方法的源码，如下所示。

```java
public void execute(Runnable command) {
    //如果提交的任务为空，则抛出空指针异常
    if (command == null)
        throw new NullPointerException();
    //获取线程池的状态和线程池中线程的数量
    int c = ctl.get();
    //线程池中的线程数量小于corePoolSize的值
    if (workerCountOf(c) < corePoolSize) {
        //重新开启线程执行任务
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //如果线程池处于RUNNING状态，则将任务添加到阻塞队列中
    if (isRunning(c) && workQueue.offer(command)) {
        //再次获取线程池的状态和线程池中线程的数量，用于二次检查
        int recheck = ctl.get();
        //如果线程池没有未处于RUNNING状态，从队列中删除任务
        if (! isRunning(recheck) && remove(command))
            //执行拒绝策略
            reject(command);
        //如果线程池为空，则向线程池中添加一个线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //任务队列已满，则新增worker线程，如果新增线程失败，则执行拒绝策略
    else if (!addWorker(command, false))
        reject(command);
```

```
}
```

整个任务的执行流程，我们可以简化成下图所示。



接下来，我们拆解execute(Runnable)方法，具体分析execute(Runnable)方法的执行逻辑。

（1）线程池中的线程数是否小于corePoolSize核心线程数，如果小于corePoolSize核心线程数，则向workers工作线程集合中添加一个核心线程执行任务。代码如下所示。

```
//线程池中的线程数量小于corePoolSize的值
if (workerCountOf(c) < corePoolSize) {
    //重新开启线程执行任务
    if (addWorker(command, true))
        return;
    c = ctl.get();
}
```

（2）如果线程池中的线程数量大于corePoolSize核心线程数，则判断当前线程池是否处于RUNNING状态，如果处于RUNNING状态，则添加任务到待执行的任务队列中。注意：这里向任务队列添加任务时，需要判断线程池是否处于RUNNING状态，只有线程池处于RUNNING状态时，才能向任务队列添加新任务。否则，会执行拒绝策略。代码如下所示。

```
if (isRunning(c) && workQueue.offer(command))
```

（3）向任务队列中添加任务成功，由于其他线程可能会修改线程池的状态，所以这里需要对线程池进行二次检查，如果当前线程池的状态不再是RUNNING状态，则需要将添加的任务从任务队列中移除，执行后续的拒绝策略。如果当前线程池仍然处于RUNNING状态，则判断线程池是否为空，如果线程池中不存在任何线程，则新建一个线程添加到线程池中，如下所示。

```
//再次获取线程池的状态和线程池中线程的数量，用于二次检查
int recheck = ctl.get();
//如果线程池没有未处于RUNNING状态，从队列中删除任务
if (! isRunning(recheck) && remove(command))
    //执行拒绝策略
    reject(command);
//如果线程池为空，则向线程池中添加一个线程
else if (workerCountOf(recheck) == 0)
    addWorker(null, false);
```

（4）如果在步骤（3）中向任务队列中添加任务失败，则尝试开启新的线程执行任务。此时，如果线程池中的线程数量已经大于线程池中的最大线程数maximumPoolSize，则不能再启动新线程。此时，表示线程池中的任务队列已满，并且线程池中的线程已满，需要执行拒绝策略，代码如下所示。

```
//任务队列已满，则新增worker线程，如果新增线程失败，则执行拒绝策略
else if (!addWorker(command, false))
    reject(command);
```

这里，我们将execute(Runnable)方法拆解，结合流程图来理解线程池中任务的执行流程就比较简单了。可以这么说，execute(Runnable)方法的逻辑基本上就是一般线程池的执行逻辑，理解了execute(Runnable)方法，就基本理解了线程池的执行逻辑。

注意：有关**ScheduledThreadPoolExecutor类和ForkJoinPool类执行线程池的逻辑，在【高并发专题】系列文章中的后文中会详细说明，理解了这些类的执行逻辑，就基本全面掌握了线程池的执行流程。**

在分析execute(Runnable)方法的源码时，我们发现execute(Runnable)方法中多处调用了addWorker(Runnable, boolean)方法，接下来，我们就一起分析下addWorker(Runnable, boolean)方法的逻辑。

addWorker(Runnable, boolean)方法

总体上，addWorker(Runnable, boolean)方法可以分为三部分，第一部分是使用CAS安全的向线程池中添加工作线程；第二部分是创建新的工作线程；第三部分则是将任务通过安全的并发方式添加到workers中，并启动工作线程执行任务。

接下来，我们看下addWorker(Runnable, boolean)方法的源码，如下所示。

```
private boolean addWorker(Runnable firstTask, boolean core) {
    //标记重试的标识
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // 检查队列是否在某些特定的条件下为空
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;
        //下面循环的主要作用为通过CAS方式增加线程的个数
        for (;;) {
            //获取线程池中的线程数量
            int wc = workerCountOf(c);
            //如果线程池中的线程数量超出限制，直接返回false
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            //通过CAS方式向线程池新增线程数量
            if (compareAndIncrementWorkerCount(c))
                //通过CAS方式保证只有一个线程执行成功，跳出最外层循环
                break retry;
            //重新获取ctl的值
            c = ctl.get();
            //如果CAS操作失败了，则需要在内循环中重新尝试通过CAS新增线程数量
            if (runStateOf(c) != rs)
                continue retry;
        }
    }

    //跳出最外层for循环，说明通过CAS新增线程数量成功
    //此时创建新的工作线程
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        //将执行的任务封装成worker
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            //独占锁，保证操作workers时的同步
            final ReentrantLock mainLock = this.mainLock;
```

```
            mainLock.lock();
            try {
                //此处需要重新检查线程池状态
                //原因是在获得锁之前可能其他的线程改变了线程池的状态
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive())
                        throw new IllegalThreadStateException();
                    //向worker中添加新任务
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    //将是否添加了新任务的标识设置为true
                    workerAdded = true;
                }
            } finally {
                //释放独占锁
                mainLock.unlock();
            }
            //添加新任成功，则启动线程执行任务
            if (workerAdded) {
                t.start();
                //将任务是否已经启动的标识设置为true
                workerStarted = true;
            }
        }
    } finally {
        //如果任务未启动或启动失败，则调用addWorkerFailed(Worker)方法
        if (! workerStarted)
            addWorkerFailed(w);
    }
    //返回是否启动任务的标识
    return workerStarted;
}
```

乍一看，addWorker(Runnable, boolean)方法还蛮长的，这里，我们还是将addWorker(Runnable, boolean)方法进行拆解。

（1）检查任务队列是否在某些特定的条件下为空，代码如下所示。

```
// 检查队列是否在某些特定的条件下为空
if (rs >= SHUTDOWN &&
    ! (rs == SHUTDOWN &&
       firstTask == null &&
       ! workQueue.isEmpty()))
    return false;
```

（2）在通过步骤（1）的校验后，则进入内层for循环，在内层for循环中通过CAS来增加线程池中的线程数量，如果CAS操作成功，则直接退出双重for循环。如果CAS操作失败，则查看当前线程池的状态是否发生了变化，如果线程池的状态发生了变化，则通过continue关键字重新通过外层for循环校验任务队列，检验通过再次执行内层for循环的CAS操作。如果线程池的状态没有发生变化，此时上一次CAS操作失败了，则继续尝试CAS操作。代码如下所示。

```
for (;;) {
    //获取线程池中的线程数量
    int wc = workerCountOf(c);
    //如果线程池中的线程数量超出限制，直接返回false
    if (wc >= CAPACITY ||
        wc >= (core ? corePoolSize : maximumPoolSize))
        return false;
    //通过CAS方式向线程池新增线程数量
    if (compareAndIncrementWorkerCount(c))
        //通过CAS方式保证只有一个线程执行成功，跳出最外层循环
        break retry;
    //重新获取ctl的值
    c = ctl.get();
    //如果CAS操作失败了，则需要在内循环中重新尝试通过CAS新增线程数量
    if (runStateOf(c) != rs)
        continue retry;
```

```
}
```

（3）CAS操作成功后，表示向线程池中成功添加了工作线程，此时，还没有线程去执行任务。使用全局的独占锁mainLock来将新增的工作线程Worker对象安全的添加到workers中。

总体逻辑就是：创建新的Worker对象，并获取Worker对象中的执行线程，如果线程不为空，则获取独占锁，获取锁成功后，再次检查线线程的状态，这是避免在获取独占锁之前其他线程修改了线程池的状态，或者关闭了线程池。如果线程池关闭，则需要释放锁。否则将新增加的线程添加到工作集合中，释放锁并启动线程执行任务。将是否启动线程的标识设置为true。最后，判断线程是否启动，如果没有启动，则调用addWorkerFailed(Worker)方法。最终返回线程是否起送的标识。

```
//跳出最外层for循环，说明通过CAS新增线程数量成功
//此时创建新的工作线程
boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    //将执行的任务封装成worker
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        //独占锁，保证操作workers时的同步
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //此处需要重新检查线程池状态
            //原因是在获得锁之前可能其他的线程改变了线程池的状态
            int rs = runStateOf(ctl.get());

            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive())
                    throw new IllegalThreadStateException();
                //向worker中添加新任务
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                //将是否添加了新任务的标识设置为true
                workerAdded = true;
            }
        } finally {
            //释放独占锁
            mainLock.unlock();
        }
        //添加新任成功，则启动线程执行任务
        if (workerAdded) {
            t.start();
            //将任务是否已经启动的标识设置为true
            workerStarted = true;
        }
    }
} finally {
    //如果任务未启动或启动失败，则调用addWorkerFailed(Worker)方法
    if (! workerStarted)
        addWorkerFailed(w);
}
//返回是否启动任务的标识
return workerStarted;
```

addWorkerFailed(Worker)方法

在addWorker(Runnable, boolean)方法中，如果添加工作线程失败或者工作线程启动失败时，则会调用addWorkerFailed(Worker)方法，下面我们就来看看addWorkerFailed(Worker)方法的实现，如下所示。

```
private void addWorkerFailed(Worker w) {
    //获取独占锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
```

```
        //如果Worker任务不为空
        if (w != null)
            //将任务从workers集合中移除
            workers.remove(w);
        //通过CAS将任务数量减1
        decrementWorkerCount();
        tryTerminate();
    } finally {
        //释放锁
        mainLock.unlock();
    }
}
```

addWorkerFailed(Worker)方法的逻辑就比较简单了，获取独占锁，将任务从workers中移除，并且通过CAS将任务的数量减1，最后释放锁。

拒绝策略

我们在分析execute(Runnable)方法时，线程池会在适当的时候调用reject(Runnable)方法来执行相应的拒绝策略，我们看下reject(Runnable)方法的实现，如下所示。

```
final void reject(Runnable command) {
    handler.rejectedExecution(command, this);
}
```

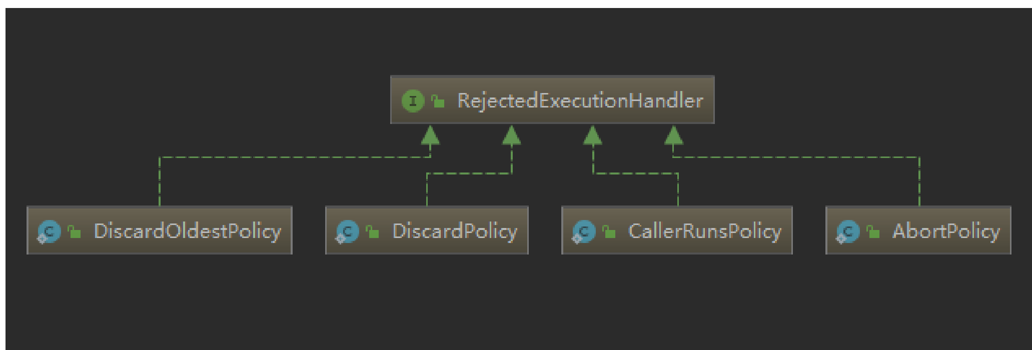
通过代码，我们发现调用的是handler的rejectedExecution方法，handler又是个什么鬼，我们继续跟进代码，如下所示。

```
private volatile RejectedExecutionHandler handler;
```

再看看RejectedExecutionHandler是个啥类型，如下所示。

```
package java.util.concurrent;

public interface RejectedExecutionHandler {

    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

可以发现RejectedExecutionHandler是个接口，定义了一个rejectedExecution(Runnable, ThreadPoolExecutor)方法。既然RejectedExecutionHandler是个接口，那我们就看看有哪些类实现了RejectedExecutionHandler接口。



看到这里，我们发现RejectedExecutionHandler接口的实现类正是线程池默认提供的四种拒绝策略的实现类。

至于reject(Runnable)方法中具体会执行哪个类的拒绝策略，是根据创建线程池时传递的参数决定的。如果没有传递拒绝策略，则默认会执行AbortPolicy类的拒绝策略。否则会执行传递的类的拒绝策略。

在创建线程池时，除了能够传递JDK默认提供的拒绝策略外，还可以传递自定义的拒绝策略。如果想使用自定义的拒绝策略，则只需要实现RejectedExecutionHandler接口，并重写rejectedExecution(Runnable, ThreadPoolExecutor)方法即可。例如，下面的代码。

```java
public class CustomPolicy implements RejectedExecutionHandler {

    public CustomPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            System.out.println("使用调用者所在的线程来执行任务")
            r.run();
        }
    }
}
```

使用如下方式创建线程池。

```java
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                       60L, TimeUnit.SECONDS,
                       new SynchronousQueue<Runnable>(),
                       Executors.defaultThreadFactory(),
               new CustomPolicy());
```

至此，线程池执行任务的整体核心逻辑分析结束。

通过源码深度分析线程池中Worker线程的执行流程

在《高并发之——通过ThreadPoolExecutor类的源码深度解析线程池执行任务的核心流程》一节中我们深度分析了线程池执行任务的核心流程，在ThreadPoolExecutor类的addWorker(Runnable, boolean)方法中，使用CAS安全的更新线程的数量之后，接下来就是创建新的Worker线程执行任务，所以，我们先来分析下Worker类的源码。

Worker类分析

Worker类从类的结构上来看，继承了AQS（AbstractQueuedSynchronizer类）并实现了Runnable接口。本质上，Worker类既是一个同步组件，也是一个执行任务的线程。接下来，我们看下Worker类的源码，如下所示。

```java
private final class Worker extends AbstractQueuedSynchronizer implements Runnable {
    private static final long serialVersionUID = 6138294804551838833L;
    //执行任务的线程类
    final Thread thread;
    //初始化执行的任务，第一次执行的任务
    Runnable firstTask;
    //完成任务的计数
    volatile long completedTasks;
    //Worker类的构造方法，初始化任务并调用线程工厂创建执行任务的线程
    Worker(Runnable firstTask) {
        setState(-1);
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }
    //重写Runnable接口的run()方法
    public void run() {
        //调用ThreadPoolExecutor类的runWorker(Worker)方法
        runWorker(this);
    }

    //检测是否是否获取到锁
    //state=0表示未获取到锁
    //state=1表示已获取到锁
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }

    //使用AQS设置线程状态
    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
```

```
        return false;
    }

    //尝试释放锁
    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }

    void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}
```

在Worker类的构造方法中，可以看出，首先将同步状态state设置为-1，设置为-1是为了防止runWorker方法运行之前被中断。这是因为如果其他线程调用线程池的shutdownNow()方法时，如果Worker类中的state状态的值大于0，则会中断线程，如果state状态的值为-1，则不会中断线程。

Worker类实现了Runnable接口，需要重写run方法，而Worker的run方法本质上调用的是ThreadPoolExecutor类的runWorker方法，在runWorker方法中，会首先调用unlock方法，该方法会将state置为0，所以这个时候调用shutDownNow方法就会中断当前线程，而这个时候已经进入了runWork方法，就不会在还没有执行runWorker方法的时候就中断线程。

注意：大家需要重点理解Worker类的实现。

Worker类中调用了ThreadPoolExecutor类的runWorker(Worker)方法。接下来，我们一起看下ThreadPoolExecutor类的runWorker(Worker)方法的实现。

runWorker(Worker)方法

首先，我们看下RunWorker(Worker)方法的源码，如下所示。

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    //释放锁，将state设置为0,允许中断任务的执行
    w.unlock();
    boolean completedAbruptly = true;
    try {
        //如果任务不为空，或者从任务队列中获取的任务不为空，则执行while循环
        while (task != null || (task = getTask()) != null) {
            //如果任务不为空，则获取Worker工作线程的独占锁
            w.lock();
            //如果线程已经停止，或者中断线程后线程终止并且没有成功中断线程
            //大家好好理解下这个逻辑
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                //中断线程
                wt.interrupt();
            try {
                //执行任务前执行的逻辑
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    //调用Runable接口的run方法执行任务
                    task.run();
                } catch (RuntimeException x) {
```

```
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    //执行任务后执行的逻辑
                    afterExecute(task, thrown);
                }
            } finally {
                //任务执行完成后，将其设置为空
                task = null;
                //完成的任务数量加1
                w.completedTasks++;
                //释放工作线程获得的锁
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        //执行退出Worker线程的逻辑
        processWorkerExit(w, completedAbruptly);
    }
}
```

这里，我们拆解runWorker(Worker)方法。

（1）获取当前线程的句柄和工作线程中的任务，并将工作线程中的任务设置为空，执行unlock方法释放锁，将state状态设置为0，此时可以中断工作线程，代码如下所示。

```
Thread wt = Thread.currentThread();
Runnable task = w.firstTask;
w.firstTask = null;
//释放锁，将state设置为0,允许中断任务的执行
w.unlock();
```

（2）在while循环中进行判断，如果任务不为空，或者从任务队列中获取的任务不为空，则执行while循环，否则，调用processWorkerExit(Worker, boolean)方法退出Worker工作线程。

```
while (task != null || (task = getTask()) != null)
```

（3）如果满足while的循环条件，首先获取工作线程内部的独占锁，并执行一系列的逻辑判断来检测是否需要中断当前线程的执行，代码如下所示。

```
//如果任务不为空，则获取Worker工作线程的独占锁
w.lock();
//如果线程已经停止，或者中断线程后线程终止并且没有成功中断线程
//大家好好理解下这个逻辑
if ((runStateAtLeast(ctl.get(), STOP) ||
     (Thread.interrupted() &&
      runStateAtLeast(ctl.get(), STOP))) &&
    !wt.isInterrupted())
    //中断线程
    wt.interrupt();
```

（4）调用执行任务前执行的逻辑，如下所示

```
//执行任务前执行的逻辑
beforeExecute(wt, task);
```

（5）调用Runable接口的run方法执行任务

```
//调用Runable接口的run方法执行任务
task.run();
```

（6）调用执行任务后执行的逻辑

```
//执行任务后执行的逻辑
afterExecute(task, thrown);
```

（7）将完成的任务设置为空，完成的任务数量加1并释放工作线程的锁。

```
//任务执行完成后，将其设置为空
task = null;
//完成的任务数量加1
w.completedTasks++;
//释放工作线程获得的锁
w.unlock();
```

（8）退出Worker线程的执行，如下所示

```
//执行退出Worker线程的逻辑
processWorkerExit(w, completedAbruptly);
```

从代码分析上可以看到，当从Worker线程中获取的任务为空时，会调用getTask()方法从任务队列中获取任务，接下来，我们看下getTask()方法的实现。

getTask()方法

我们先来看下getTask()方法的源代码，如下所示。

```
private Runnable getTask() {
    //轮询是否超时的标识
    boolean timedOut = false;
    //自旋for循环
    for (;;) {
        //获取ctl
        int c = ctl.get();
        //获取线程池的状态
        int rs = runStateOf(c);

        //检测任务队列是否在线程池停止或关闭的时候为空
        //也就是说任务队列是否在线程池未正常运行时为空
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            //减少Worker线程的数量
            decrementWorkerCount();
            return null;
        }
        //获取线程池中线程的数量
        int wc = workerCountOf(c);

        //检测当前线程池中的线程数量是否大于corePoolSize的值或者是否正在等待执行任务
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        //如果线程池中的线程数量大于corePoolSize
        //获取大于corePoolSize或者是否正在等待执行任务并且轮询超时
        //并且当前线程池中的线程数量大于1或者任务队列为空
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            //成功减少线程池中的工作线程数量
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            //从任务队列中获取任务
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            //任务不为空直接返回任务
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
```

```
            timedOut = false;
        }
    }
}
```

getTask()方法的逻辑比较简单，大家看源码就可以了，我这里就不重复描述了。

接下来，我们看下在正式调用Runnable的run()方法前后，执行的beforeExecute方法和afterExecute方法。

beforeExecute(Thread, Runnable)方法

beforeExecute(Thread, Runnable)方法的源代码如下所示。

```
protected void beforeExecute(Thread t, Runnable r) { }
```

可以看到，beforeExecute(Thread, Runnable)方法的方法体为空，我们可以创建ThreadPoolExecutor的子类来重写beforeExecute(Thread, Runnable)方法，使得线程池正式执行任务之前，执行我们自己定义的业务逻辑。

afterExecute(Runnable, Throwable)方法

afterExecute(Runnable, Throwable)方法的源代码如下所示。

```
protected void afterExecute(Runnable r, Throwable t) { }
```

可以看到，afterExecute(Runnable, Throwable)方法的方法体同样为空，我们可以创建ThreadPoolExecutor的子类来重写afterExecute(Runnable, Throwable)方法，使得线程池在执行任务之后执行我们自己定义的业务逻辑。

接下来，就是退出工作线程的processWorkerExit(Worker, boolean)方法。

processWorkerExit(Worker, boolean)方法

processWorkerExit(Worker, boolean)方法的逻辑主要是执行退出Worker线程，并且对一些资源进行清理，源代码如下所示。

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    //执行过程中出现了异常，突然中断
    if (completedAbruptly)
        //将工作线程的数量减1
        decrementWorkerCount();
    //获取全局锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //累加完成的任务数量
        completedTaskCount += w.completedTasks;
        //将完成的任务从workers集合中移除
        workers.remove(w);
    } finally {
        //释放锁
        mainLock.unlock();
    }
    //尝试终止工作线程的执行
    tryTerminate();
    //获取ctl
    int c = ctl.get();
    //判断当前线程池的状态是否小于STOP（RUNNING或者SHUTDOWN）
    if (runStateLessThan(c, STOP)) {
        //如果没有突然中断完成
        if (!completedAbruptly) {
            //如果allowCoreThreadTimeOut为true，为min赋值为0，否则赋值为corePoolSize
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            //如果min为0并且工作队列不为空
            if (min == 0 && ! workQueue.isEmpty())
                //min的值设置为1
                min = 1;
            //如果线程池中的线程数量大于min的值
            if (workerCountOf(c) >= min)
                //返回，不再执行程序
                return;
        }
```

```
            //调用addWorker方法
            addWorker(null, false);
        }
    }
```

接下来，我们拆解processWorkerExit(Worker, boolean)方法。

（1）执行过程中出现了异常，突然中断执行，则将工作线程数量减1，如下所示。

```
//执行过程中出现了异常，突然中断
if (completedAbruptly)
    //将工作线程的数量减1
    decrementWorkerCount();
```

（2）获取锁累加完成的任务数量，并将完成的任务从workers集合中移除，并释放，如下所示。

```
//获取全局锁
final ReentrantLock mainLock = this.mainLock;
mainLock.lock();
try {
    //累加完成的任务数量
    completedTaskCount += w.completedTasks;
    //将完成的任务从workers集合中移除
    workers.remove(w);
} finally {
    //释放锁
    mainLock.unlock();
}
```

（3）尝试终止工作线程的执行

```
//尝试终止工作线程的执行
tryTerminate();
```

（4）处判断当前线程池中的线程个数是否小于核心线程数，如果是，需要新增一个线程保证有足够的线程可以执行任务队列中的任务或者提交的任务。

```
//获取ctl
int c = ctl.get();
//判断当前线程池的状态是否小于STOP（RUNNING或者SHUTDOWN）
if (runStateLessThan(c, STOP)) {
    //如果没有突然中断完成
    if (!completedAbruptly) {
        //如果allowCoreThreadTimeOut为true，为min赋值为0，否则赋值为corePoolSize
        int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
        //如果min为0并且工作队列不为空
        if (min == 0 && ! workQueue.isEmpty())
            //min的值设置为1
            min = 1;
        //如果线程池中的线程数量大于min的值
        if (workerCountOf(c) >= min)
            //返回，不再执行程序
            return;
    }
    //调用addWorker方法
    addWorker(null, false);
}
```

接下来，我们看下tryTerminate()方法。

tryTerminate()方法

tryTerminate()方法的源代码如下所示。

```
final void tryTerminate() {
    //自旋for循环
    for (;;) {
        //获取ctl
```

```
        int c = ctl.get();
        //如果线程池的状态为RUNNING
        //或者状态大于TIDYING
        //或者状态为SHUTDOWN并且任务队列为空
        //直接返回程序，不再执行后续逻辑
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;
        //如果当前线程池中的线程数量不等于0
        if (workerCountOf(c) != 0) {
            //中断线程的执行
            interruptIdleWorkers(ONLY_ONE);
            return;
        }
        //获取线程池的全局锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //通过CAS将线程池的状态设置为TIDYING
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    //调用terminated()方法
                    terminated();
                } finally {
                    //将线程池状态设置为TERMINATED
                    ctl.set(ctlOf(TERMINATED, 0));
                    //唤醒所有因为调用线程池的awaitTermination方法而被阻塞的线程
                    termination.signalAll();
                }
                return;
            }
        } finally {
            //释放锁
            mainLock.unlock();
        }
    }
}
```

（1）获取ctl，根据情况设置线程池状态或者中断线程的执行，并返回。

```
//获取ctl
int c = ctl.get();
//如果线程池的状态为RUNNING
//或者状态大于TIDYING
//或者状态为SHUTDOWN并且任务队列为空
//直接返回程序，不再执行后续逻辑
if (isRunning(c) ||
    runStateAtLeast(c, TIDYING) ||
    (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
    return;
//如果当前线程池中的线程数量不等于0
if (workerCountOf(c) != 0) {
    //中断线程的执行
    interruptIdleWorkers(ONLY_ONE);
    return;
}
```

（2）获取全局锁，通过CAS设置线程池的状态，调用terminated()方法执行逻辑，最终将线程池的状态设置为TERMINATED，唤醒所有因为调用线程池的awaitTermination方法而被阻塞的线程，最终释放锁，如下所示。

```
//获取线程池的全局
final ReentrantLock mainLock = this.mainLock;
mainLock.lock();
try {
    //通过CAS将线程池的状态设置为TIDYING
    if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
        try {
            //调用terminated()方法
```

```java
                terminated();
            } finally {
                //将线程池状态设置为TERMINATED
                ctl.set(ctlOf(TERMINATED, 0));
                //唤醒所有因为调用线程池的awaitTermination方法而被阻塞的线程
                termination.signalAll();
            }
            return;
        }
    } finally {
        //释放锁
        mainLock.unlock();
    }
```

接下来，看下terminated()方法。

terminated()方法

terminated()方法的源代码如下所示。

```java
protected void terminated() { }
```

可以看到，terminated()方法的方法体为空，我们可以创建ThreadPoolExecutor的子类来重写terminated()方法，值得Worker线程调用tryTerminate()方法时执行我们自己定义的terminated()方法的业务逻辑。

从源码角度深度解析线程池是如何实现优雅退出的

本文，我们就来从源码角度深度解析线程池是如何优雅的退出程序的。首先，我们来看下ThreadPoolExecutor类中的shutdown()方法。

shutdown()方法

当使用线程池的时候，调用了shutdown()方法后，线程池就不会再接受新的执行任务了。但是在调用shutdown()方法之前放入任务队列中的任务还是要执行的。此方法是非阻塞方法，调用后会立即返回，并不会等待任务队列中的任务全部执行完毕后再返回。我们看下shutdown()方法的源代码，如下所示。

```java
public void shutdown() {
    //获取线程池的全局锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //检查是否有关闭线程池的权限
        checkShutdownAccess();
        //将当前线程池的状态设置为SHUTDOWN
        advanceRunState(SHUTDOWN);
        //中断Worker线程
        interruptIdleWorkers();
        //为ScheduledThreadPoolExecutor调用钩子函数
        onShutdown(); // hook for
    } finally {
        //释放线程池的全局锁
        mainLock.unlock();
    }
    //尝试将状态变为TERMINATED
    tryTerminate();
}
```

总体来说，shutdown()方法的代码比较简单，首先检查了是否有权限来关闭线程池，如果有权限，则再次检测是否有中断工作线程的权限，如果没有权限，则会抛出SecurityException异常，代码如下所示。

```java
//检查是否有关闭线程池的权限
checkShutdownAccess();
//将当前线程池的状态设置为SHUTDOWN
advanceRunState(SHUTDOWN);
//中断Worker线程
interruptIdleWorkers();
```

其中，checkShutdownAccess()方法的实现代码如下所示。

```
private void checkShutdownAccess() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkPermission(shutdownPerm);
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                security.checkAccess(w.thread);
        } finally {
            mainLock.unlock();
        }
    }
}
```

对于checkShutdownAccess()方法的代码理解起来比较简单，就是检测是否具有关闭线程池的权限，期间使用了线程池的全局锁。

接下来，我们看advanceRunState(int)方法的源代码，如下所示。

```
private void advanceRunState(int targetState) {
    for (;;) {
        int c = ctl.get();
        if (runStateAtLeast(c, targetState) ||
            ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
            break;
    }
}
```

advanceRunState(int)方法的整体逻辑就是：判断当前线程池的状态是否为指定的状态，在shutdown()方法中传递的状态是SHUTDOWN，如果是SHUTDOWN，则直接返回；如果不是SHUTDOWN，则将当前线程池的状态设置为SHUTDOWN。

接下来，我们看看showdown()方法调用的interruptIdleWorkers()方法，如下所示。

```
private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}
```

可以看到，interruptIdleWorkers()方法调用的是interruptIdleWorkers(boolean)方法，继续看interruptIdleWorkers(boolean)方法的源代码，如下所示。

```
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}
```

上述代码的总体逻辑为：获取线程池的全局锁，循环所有的工作线程，检测线程是否被中断，如果没有被中断，并且Worker线程获得了锁，则执行线程的中断方法，并释放线程获取到的锁。此时如果onlyOne参数为true，则退出循环。否则，循环所有的工作线程，执行相同的操作。最终，释放线程池的全局锁。

接下来，我们看下shutdownNow()方法。

shutdownNow()方法

如果调用了线程池的shutdownNow()方法，则线程池不会再接受新的执行任务，也会将任务队列中存在的任务丢弃，正在执行的Worker线程也会被立即中断，同时，方法会立刻返回，此方法存在一个返回值，也就是当前任务队列中被丢弃的任务列表。

shutdownNow()方法的源代码如下所示。

```
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //检查是否有关闭权限
        checkShutdownAccess();
        //设置线程池的状态为STOP
        advanceRunState(STOP);
        //中断所有的Worker线程
        interruptWorkers();
        //将任务队列中的任务移动到tasks集合中
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
    /尝试将状态变为TERMINATED
    tryTerminate();
    //返回tasks集合
    return tasks;
}
```

shutdownNow()方法的源代码的总体逻辑与shutdown()方法基本相同，只是shutdownNow()方法将线程池的状态设置为STOP，中断所有的Worker线程，并且将任务队列中的所有任务移动到tasks集合中并返回。

可以看到，shutdownNow()方法中断所有的线程时，调用了interruptWorkers()方法，接下来，我们就看下interruptWorkers()方法的源代码，如下所示。

```
private void interruptWorkers() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers)
            w.interruptIfStarted();
    } finally {
        mainLock.unlock();
    }
}
```

interruptWorkers()方法的逻辑比较简单，就是获得线程池的全局锁，循环所有的工作线程，依次中断线程，最后释放线程池的全局锁。

在interruptWorkers()方法的内部，实际上调用的是Worker类的interruptIfStarted()方法来中断线程，我们看下Worker类的interruptIfStarted()方法的源代码，如下所示。

```
void interruptIfStarted() {
    Thread t;
    if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
        try {
            t.interrupt();
        } catch (SecurityException ignore) {
        }
    }
}
```

发现其本质上调用的还是Thread类的interrupt()方法来中断线程。

awaitTermination(long, TimeUnit)方法

当线程池调用了awaitTermination(long, TimeUnit)方法后，会阻塞调用者所在的线程，直到线程池的状态修改为TERMINATED才返回，或者达到了超时时间返回。接下来，我们看下awaitTermination(long, TimeUnit)方法的源代码，如下所示。

```java
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
    //获取距离超时时间剩余的时长
    long nanos = unit.toNanos(timeout);
    //获取Worker线程的的全局锁
    final ReentrantLock mainLock = this.mainLock;
    //加锁
    mainLock.lock();
    try {
        for (;;) {
            //当前线程池状态为TERMINATED状态，会返回true
            if (runStateAtLeast(ctl.get(), TERMINATED))
                return true;
            //达到超时时间，已超时，则返回false
            if (nanos <= 0)
                return false;
            //重置距离超时时间的剩余时长
            nanos = termination.awaitNanos(nanos);
        }
    } finally {
        //释放锁
        mainLock.unlock();
    }
}
```

上述代码的总体逻辑为：首先获取Worker线程的独占锁，后在循环判断当前线程池是否已经是TERMINATED状态，如果是则直接返回true，否则检测是否已经超时，如果已经超时，则返回false。如果未超时，则重置距离超时时间的剩余时长。接下来，进入下一轮循环，再次检测当前线程池是否已经是TERMINATED状态，如果是则直接返回true，否则检测是否已经超时，如果已经超时，则返回false。如果未超时，则重置距离超时时间的剩余时长。以此循环，直到线程池的状态变为TERMINATED或者已经超时。

深入理解ScheduledThreadPoolExecutor与Timer的区别和简单示例

JDK 1.5开始提供ScheduledThreadPoolExecutor类，ScheduledThreadPoolExecutor类继承ThreadPoolExecutor类重用线程池实现了任务的周期性调度功能。在JDK 1.5之前，实现任务的周期性调度主要使用的是Timer类和TimerTask类。本文，就简单介绍下ScheduledThreadPoolExecutor类与Timer类的区别，ScheduledThreadPoolExecutor类相比于Timer类来说，究竟有哪些优势，以及二者分别实现任务调度的简单示例。

二者的区别

线程角度

- Timer是单线程模式，如果某个TimerTask任务的执行时间比较久，会影响到其他任务的调度执行。
- ScheduledThreadPoolExecutor是多线程模式，并且重用线程池，某个ScheduledFutureTask任务执行的时间比较久，不会影响到其他任务的调度执行。

系统时间敏感度

- Timer调度是基于操作系统的绝对时间的，对操作系统的时间敏感，一旦操作系统的时间改变，则Timer的调度不再精确。
- ScheduledThreadPoolExecutor调度是基于相对时间的，不受操作系统时间改变的影响。

是否捕获异常

- Timer不会捕获TimerTask抛出的异常，加上Timer又是单线程的。一旦某个调度任务出现异常，则整个线程就会终止，其他需要调度的任务也不再执行。
- ScheduledThreadPoolExecutor基于线程池来实现调度功能，某个任务抛出异常后，其他任务仍能正常执行。

任务是否具备优先级

- Timer中执行的TimerTask任务整体上没有优先级的概念，只是按照系统的绝对时间来执行任务。
- ScheduledThreadPoolExecutor中执行的ScheduledFutureTask类实现了java.lang.Comparable接口和java.util.concurrent.Delayed接口，这也就说明了ScheduledFutureTask类中实现了两个非常重要的方法，一个是java.lang.Comparable接口的compareTo方法，一个是java.util.concurrent.Delayed接口的getDelay方法。在ScheduledFutureTask类中compareTo方法方法实现了任务的比较，距离下次执行的时间间隔短的任务会排在前面，也就是说，距离下次执行的时间间隔短的任务的优先级比较高。而getDelay方法则能够返回距离下次任务执行的时间间隔。

是否支持对任务排序

- Timer不支持对任务的排序。
- ScheduledThreadPoolExecutor类中定义了一个静态内部类DelayedWorkQueue，DelayedWorkQueue类本质上是一个有序队列，为需要调度的每个任务按照距离下次执行时间间隔的大小来排序

能否获取返回的结果

- Timer中执行的TimerTask类只是实现了java.lang.Runnable接口，无法从TimerTask中获取返回的结果。
- ScheduledThreadPoolExecutor中执行的ScheduledFutureTask类继承了FutureTask类，能够通过Future来获取返回的结果。

通过以上对ScheduledThreadPoolExecutor类和Timer类的分析对比，相信在JDK 1.5之后，就没有使用Timer来实现定时任务调度的必要了。

二者简单的示例

这里，给出使用Timer和ScheduledThreadPoolExecutor实现定时调度的简单示例，为了简便，我这里就直接使用匿名内部类的形式来提交任务。

Timer类简单示例

源代码示例如下所示。

```
package io.binghe.concurrent.lab09;

import java.util.Timer;
import java.util.TimerTask;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试Timer
 */
public class TimerTest {

    public static void main(String[] args) throws InterruptedException {
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                System.out.println("测试Timer类");
            }
        }, 1000, 1000);
        Thread.sleep(10000);
        timer.cancel();
    }
}
```

运行结果如下所示。

```
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
测试Timer类
```

ScheduledThreadPoolExecutor类简单示例

源代码示例如下所示。

```
package io.binghe.concurrent.lab09;

import java.util.concurrent.*;

/**
 * @author binghe
```

```
 * @version 1.0.0
 * @description 测试ScheduledThreadPoolExecutor
 */
public class ScheduledThreadPoolExecutorTest {
    public static void main(String[] args) throws  InterruptedException {
        ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(3);
        scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                System.out.println("测试测试ScheduledThreadPoolExecutor");
            }
        }, 1, 1, TimeUnit.SECONDS);

        //主线程休眠10秒
        Thread.sleep(10000);

        System.out.println("正在关闭线程池...");
        // 关闭线程池
        scheduledExecutorService.shutdown();
        boolean isClosed;
        // 等待线程池终止
        do {
            isClosed = scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
            System.out.println("正在等待线程池中的任务执行完成");
        } while(!isClosed);

        System.out.println("所有线程执行结束，线程池关闭");
    }
}
```

运行结果如下所示。

```
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
测试测试ScheduledThreadPoolExecutor
正在关闭线程池...
测试测试ScheduledThreadPoolExecutor
正在等待线程池中的任务执行完成
所有线程执行结束，线程池关闭
```

注意：关于Timer和ScheduledThreadPoolExecutor还有其他的使用方法，这里，我就简单列出以上两个使用示例，更多的使用方法大家可以自行实现。

深度解析ScheduledThreadPoolExecutor类的源代码

在【高并发专题】的专栏中，我们深度分析了ThreadPoolExecutor类的源代码，而ScheduledThreadPoolExecutor类是ThreadPoolExecutor类的子类。今天我们就来一起手撕ScheduledThreadPoolExecutor类的源代码。

构造方法

我们先来看下ScheduledThreadPoolExecutor的构造方法，源代码如下所示。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue());
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), threadFactory);
}
```

```java
public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), handler);
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory,
RejectedExecutionHandler handler) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue(), threadFactory, handler);
}
```

从代码结构上来看，ScheduledThreadPoolExecutor类是ThreadPoolExecutor类的子类，ScheduledThreadPoolExecutor类的构造方法实际上调用的是ThreadPoolExecutor类的构造方法。

schedule方法

接下来，我们看一下ScheduledThreadPoolExecutor类的schedule方法，源代码如下所示。

```java
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) {
    //如果传递的Runnable对象和TimeUnit时间单位为空
    //抛出空指针异常
    if (command == null || unit == null)
        throw new NullPointerException();
    //封装任务对象，在decorateTask方法中直接返回ScheduledFutureTask对象
    RunnableScheduledFuture<?> t = decorateTask(command, new ScheduledFutureTask<Void>(command, null,
triggerTime(delay, unit)));
    //执行延时任务
    delayedExecute(t);
    //返回任务
    return t;
}

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)
    //如果传递的Callable对象和TimeUnit时间单位为空
    //抛出空指针异常
    if (callable == null || unit == null)
        throw new NullPointerException();
    //封装任务对象，在decorateTask方法中直接返回ScheduledFutureTask对象
    RunnableScheduledFuture<V> t = decorateTask(callable,
        new ScheduledFutureTask<V>(callable, triggerTime(delay, unit)));
    //执行延时任务
    delayedExecute(t);
    //返回任务
    return t;
}
```

从源代码可以看出，ScheduledThreadPoolExecutor类提供了两个重载的schedule方法，两个schedule方法的第一个参数不同。可以传递Runnable接口对象，也可以传递Callable接口对象。在方法内部，会将Runnable接口对象和Callable接口对象封装成RunnableScheduledFuture对象，本质上就是封装成ScheduledFutureTask对象。并通过delayedExecute方法来执行延时任务。

在源代码中，我们看到两个schedule都调用了decorateTask方法，接下来，我们就看看decorateTask方法。

decorateTask方法

decorateTask方法源代码如下所示。

```java
protected <V> RunnableScheduledFuture<V> decorateTask(Runnable runnable, RunnableScheduledFuture<V>
task) {
    return task;
}

protected <V> RunnableScheduledFuture<V> decorateTask(Callable<V> callable, RunnableScheduledFuture<V>
task) {
    return task;
}
```

通过源码可以看出decorateTask方法的实现比较简单，接收一个Runnable接口对象或者Callable接口对象和封装的RunnableScheduledFuture任务，两个方法都是将RunnableScheduledFuture任务直接返回。在ScheduledThreadPoolExecutor类的子类中可以重写这两个方法。

接下来，我们继续看下scheduleAtFixedRate方法。

scheduleAtFixedRate方法

scheduleAtFixedRate方法源代码如下所示。

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period,
TimeUnit unit) {
    //传入的Runnable对象和TimeUnit为空，则抛出空指针异常
    if (command == null || unit == null)
        throw new NullPointerException();
    //如果执行周期period传入的数值小于或者等于0
    //抛出非法参数异常
    if (period <= 0)
        throw new IllegalArgumentException();
    //将Runnable对象封装成ScheduledFutureTask任务，
    //并设置执行周期
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command, null, triggerTime(initialDelay, unit),
unit.toNanos(period));
    //调用decorateTask方法，本质上还是直接返回ScheduledFutureTask对象
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    //设置执行的任务
    sft.outerTask = t;
    //执行延时任务
    delayedExecute(t);
    //返回执行的任务
    return t;
}
```

通过源码可以看出，scheduleAtFixedRate方法将传递的Runnable对象封装成ScheduledFutureTask任务对象，并设置了执行周期，下一次的执行时间相对于上一次的执行时间来说，加上了period时长，时长的具体单位由TimeUnit决定。采用固定的频率来执行定时任务。

ScheduledThreadPoolExecutor类中另一个定时调度任务的方法是scheduleWithFixedDelay方法，接下来，我们就一起看看scheduleWithFixedDelay方法。

scheduleWithFixedDelay方法

scheduleWithFixedDelay方法的源代码如下所示。

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay,
TimeUnit unit) {
    //传入的Runnable对象和TimeUnit为空，则抛出空指针异常
    if (command == null || unit == null)
        throw new NullPointerException();
    //任务延时时长小于或者等于0，则抛出非法参数异常
    if (delay <= 0)
        throw new IllegalArgumentException();
    //将Runnable对象封装成ScheduledFutureTask任务
    //并设置固定的执行周期来执行任务
    ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command, null,triggerTime(initialDelay, unit), unit.toNanos(-
delay));
    //调用decorateTask方法，本质上直接返回ScheduledFutureTask任务
    RunnableScheduledFuture<Void> t = decorateTask(command, sft);
    //设置执行的任务
    sft.outerTask = t;
    //执行延时任务
    delayedExecute(t);
    //返回任务
    return t;
}
```

从scheduleWithFixedDelay方法的源代码，我们可以看出在将Runnable对象封装成ScheduledFutureTask时，设置了执行周期，但是此时设置的执行周期与scheduleAtFixedRate方法设置的执行周期不同。此时设置的执行周期规则为：下一次任务执行的时间是上一次任务完成的时间加上delay时长，时长单位由TimeUnit决定。也就是说，具体的执行时间不是固定的，但是执行的周期是固定的，整体采用的是相对固定的延迟来执行定时任务。

如果大家细心的话，会发现在scheduleWithFixedDelay方法中设置执行周期时，传递的delay值为负数，如下所示。

```
ScheduledFutureTask<Void> sft =
        new ScheduledFutureTask<Void>(command, null, triggerTime(initialDelay, unit), unit.toNanos(-
delay));
```

这里的负数表示的是相对固定的延迟。

在ScheduledFutureTask类中，存在一个setNextRunTime方法，这个方法会在run方法执行完任务后调用，这个方法更能体现scheduleAtFixedRate方法和scheduleWithFixedDelay方法的不同，setNextRunTime方法的源码如下所示。

```
private void setNextRunTime() {
    //距离下次执行任务的时长
    long p = period;
    //固定频率执行，
    //上次执行任务的时间
    //加上任务的执行周期
    if (p > 0)
        time += p;
    //相对固定的延迟
    //使用的是系统当前时间
    //加上任务的执行周期
    else
        time = triggerTime(-p);
}
```

在setNextRunTime方法中通过对下次执行任务的时长进行判断来确定是固定频率执行还是相对固定的延迟。

triggerTime方法

在ScheduledThreadPoolExecutor类中提供了两个triggerTime方法，用于获取下一次执行任务的具体时间。triggerTime方法的源码如下所示。

```
private long triggerTime(long delay, TimeUnit unit) {
    return triggerTime(unit.toNanos((delay < 0) ? 0 : delay));
}

long triggerTime(long delay) {
    return now() +
        ((delay < (Long.MAX_VALUE >> 1)) ? delay : overflowFree(delay));
}
```

这两个triggerTime方法的代码比较简单，就是获取下一次执行任务的具体时间。有一点需要注意的是：delay < (Long.MAX_VALUE >> 1判断delay的值是否小于Long.MAX_VALUE的一半，如果小于Long.MAX_VALUE值的一半，则直接返回delay，否则需要处理溢出的情况。

我们看到在triggerTime方法中处理防止溢出的逻辑使用了overflowFree方法，接下来，我们就看看overflowFree方法的实现。

overflowFree方法

overflowFree方法的源代码如下所示。

```
private long overflowFree(long delay) {
    //获取队列中的节点
    Delayed head = (Delayed) super.getQueue().peek();
    //获取的节点不为空，则进行后续处理
    if (head != null) {
        //从队列节点中获取延迟时间
        long headDelay = head.getDelay(NANOSECONDS);
        //如果从队列中获取的延迟时间小于0，并且传递的delay
        //值减去从队列节点中获取延迟时间小于0
        if (headDelay < 0 && (delay - headDelay < 0))
            //将delay的值设置为Long.MAX_VALUE + headDelay
            delay = Long.MAX_VALUE + headDelay;
    }
    //返回延迟时间
    return delay;
}
```

通过对overflowFree方法的源码分析，可以看出overflowFree方法本质上就是为了限制队列中的所有节点的延迟时间在Long.MAX_VALUE值之内，防止在ScheduledFutureTask类中的compareTo方法中溢出。

ScheduledFutureTask类中的compareTo方法的源码如下所示。

```
public int compareTo(Delayed other) {
    if (other == this) // compare zero if same object
        return 0;
    if (other instanceof ScheduledFutureTask) {
        ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
        long diff = time - x.time;
        if (diff < 0)
            return -1;
        else if (diff > 0)
            return 1;
        else if (sequenceNumber < x.sequenceNumber)
            return -1;
        else
            return 1;
    }
    long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECONDS);
    return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;
}
```

compareTo方法的主要作用就是对各延迟任务进行排序，距离下次执行时间靠前的任务就排在前面。

delayedExecute方法

delayedExecute方法是ScheduledThreadPoolExecutor类中延迟执行任务的方法，源代码如下所示。

```
private void delayedExecute(RunnableScheduledFuture<?> task) {
    //如果当前线程池已经关闭
    //则执行线程池的拒绝策略
    if (isShutdown())
        reject(task);
    //线程池没有关闭
    else {
        //将任务添加到阻塞队列中
        super.getQueue().add(task);
        //如果当前线程池是SHUTDOWN状态
        //并且当前线程池状态下不能执行任务
        //并且成功从阻塞队列中移除任务
        if (isShutdown() &&
            !canRunInCurrentRunState(task.isPeriodic()) &&
            remove(task))
            //取消任务的执行，但不会中断执行中的任务
            task.cancel(false);
        else
            //调用ThreadPoolExecutor类中的ensurePrestart()方法
            ensurePrestart();
    }
}
```

可以看到在delayedExecute方法内部调用了canRunInCurrentRunState方法，canRunInCurrentRunState方法的源码实现如下所示。

```
boolean canRunInCurrentRunState(boolean periodic) {
    return isRunningOrShutdown(periodic ? continueExistingPeriodicTasksAfterShutdown :
executeExistingDelayedTasksAfterShutdown);
}
```

可以看到canRunInCurrentRunState方法的逻辑比较简单，就是判断线程池当前状态下能够执行任务。

另外，在delayedExecute方法内部还调用了ThreadPoolExecutor类中的ensurePrestart()方法，接下来，我们看下ThreadPoolExecutor类中的ensurePrestart()方法的实现，如下所示。

```java
void ensurePrestart() {
    int wc = workerCountOf(ctl.get());
    if (wc < corePoolSize)
        addWorker(null, true);
    else if (wc == 0)
        addWorker(null, false);
}
```

在ThreadPoolExecutor类中的ensurePrestart()方法中，首先获取当前线程池中线程的数量，如果线程数量小于corePoolSize则调用addWorker方法传递null和true，如果线程数量为0，则调用addWorker方法传递null和false。

reExecutePeriodic方法

reExecutePeriodic方法的源代码如下所示。

```java
void reExecutePeriodic(RunnableScheduledFuture<?> task) {
    //线程池当前状态下能够执行任务
    if (canRunInCurrentRunState(true)) {
        //将任务放入队列
        super.getQueue().add(task);
        //线程池当前状态下不能执行任务，并且成功移除任务
        if (!canRunInCurrentRunState(true) && remove(task))
            //取消任务
            task.cancel(false);
        else
            //调用ThreadPoolExecutor类的ensurePrestart()方法
            ensurePrestart();
    }
}
```

总体来说reExecutePeriodic方法的逻辑比较简单，但是，这里需要注意和delayedExecute方法的不同点：调用reExecutePeriodic方法的时候已经执行过一次任务，所以，并不会触发线程池的拒绝策略；传入reExecutePeriodic方法的任务一定是周期性的任务。

onShutdown方法

onShutdown方法是ThreadPoolExecutor类中的钩子函数，它是在ThreadPoolExecutor类中的shutdown方法中调用的，而在ThreadPoolExecutor类中的onShutdown方法是一个空方法，如下所示。

```java
void onShutdown() {
}
```

ThreadPoolExecutor类中的onShutdown方法交由子类实现，所以ScheduledThreadPoolExecutor类覆写了onShutdown方法，实现了具体的逻辑，ScheduledThreadPoolExecutor类中的onShutdown方法的源码实现如下所示。

```java
@Override
void onShutdown() {
    //获取队列
    BlockingQueue<Runnable> q = super.getQueue();
    //在线程池已经调用shutdown方法后，是否继续执行现有延迟任务
    boolean keepDelayed = getExecuteExistingDelayedTasksAfterShutdownPolicy();
    //在线程池已经调用shutdown方法后，是否继续执行现有定时任务
    boolean keepPeriodic = getContinueExistingPeriodicTasksAfterShutdownPolicy();
    //在线程池已经调用shutdown方法后，不继续执行现有延迟任务和定时任务
    if (!keepDelayed && !keepPeriodic) {
        //遍历队列中的所有任务
        for (Object e : q.toArray())
            //取消任务的执行
            if (e instanceof RunnableScheduledFuture<?>)
                ((RunnableScheduledFuture<?>) e).cancel(false);
        //清空队列
        q.clear();
    }
    //在线程池已经调用shutdown方法后，继续执行现有延迟任务和定时任务
    else {
        //遍历队列中的所有任务
        for (Object e : q.toArray()) {
            //当前任务是RunnableScheduledFuture类型
            if (e instanceof RunnableScheduledFuture) {
```

```java
                //将任务强转为RunnableScheduledFuture类型
                RunnableScheduledFuture<?> t = (RunnableScheduledFuture<?>)e;
                //在线程池调用shutdown方法后不继续的延迟任务或周期任务
                //则从队列中删除并取消任务
                if ((t.isPeriodic() ? !keepPeriodic : !keepDelayed) ||
                    t.isCancelled()) {
                    if (q.remove(t))
                        t.cancel(false);
                }
            }
        }
    }
    //最终调用tryTerminate()方法
    tryTerminate();
}
```

ScheduledThreadPoolExecutor类中的onShutdown方法的主要逻辑就是先判断线程池调用shutdown方法后，是否继续执行现有的延迟任务和定时任务，如果不再执行，则取消任务并清空队列；如果继续执行，将队列中的任务强转为RunnableScheduledFuture对象之后，从队列中删除并取消任务。大家需要好好理解这两种处理方式。最后调用ThreadPoolExecutor类的tryTerminate方法。

至此，ScheduledThreadPoolExecutor类中的核心方法的源代码，我们就分析完了。

深入理解Thread类源码

前言

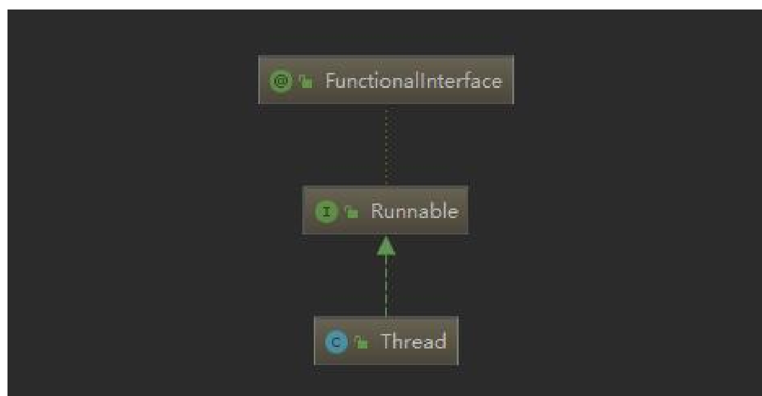
最近和一个朋友聊天，他跟我说起了他去XXX公司面试的情况，面试官的一个问题把他打懵了！竟然问他：你经常使用Thread创建线程，那你看过Thread类的源码吗？我这个朋友自然是没看过Thread类的源码，然后，就没有然后了！！！

所以，我们学习技术不仅需要知其然，更需要知其所以然，今天，我们就一起来简单看看Thread类的源码。

注意：本文是基于JDK 1.8来进行分析的。

Thread类的继承关系

我们可以使用下图来表示Thread类的继承关系。



由上图我们可以看出，Thread类实现了Runnable接口，而Runnable在JDK 1.8中被@FunctionalInterface注解标记为函数式接口，Runnable接口在JDK 1.8中的源代码如下所示。

```java
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Runnable接口的源码比较简单，只是提供了一个run()方法，这里就不再赘述了。

接下来，我们再来看看@FunctionalInterface注解的源码，如下所示。

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

可以看到，@FunctionalInterface注解声明标记在Java类上，并在程序运行时生效。

Thread类的源码剖析

Thread类定义

Thread在java.lang包下，Thread类的定义如下所示。

```
public class Thread implements Runnable {
```

加载本地资源

打开Thread类后，首先，我们会看到在Thread类的最开始部分，定义了一个静态本地方法registerNatives()，这个方法主要用来注册一些本地系统的资源。并在静态代码块中调用这个本地方法，如下所示。

```
//定义registerNatives()本地方法注册系统资源
private static native void registerNatives();
static {
    //在静态代码块中调用注册本地系统资源的方法
    registerNatives();
}
```

Thread中的成员变量

Thread类中的成员变量如下所示。

```
//当前线程的名称
private volatile String name;
//线程的优先级
private int            priority;
private Thread         threadQ;
private long           eetop;
//当前线程是否是单步线程
private boolean     single_step;
//当前线程是否在后台运行
private boolean     daemon = false;
//Java虚拟机的状态
private boolean     stillborn = false;
//真正在线程中执行的任务
private Runnable target;
//当前线程所在的线程组
private ThreadGroup group;
//当前线程的类加载器
private ClassLoader contextClassLoader;
//访问控制上下文
private AccessControlContext inheritedAccessControlContext;
//为匿名线程生成名称的编号
private static int threadInitNumber;
//与此线程相关的ThreadLocal,这个Map维护的是ThreadLocal类
ThreadLocal.ThreadLocalMap threadLocals = null;
//与此线程相关的ThreadLocal
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
//当前线程请求的堆栈大小，如果未指定堆栈大小，则会交给JVM来处理
private long stackSize;
//线程终止后存在的JVM私有状态
private long nativeParkEventPointer;
//线程的id
private long tid;
//用于生成线程id
private static long threadSeqNumber;
//当前线程的状态，初始化为0，代表当前线程还未启动
private volatile int threadStatus = 0;
//由（私有）java.util.concurrent.locks.LockSupport.setBlocker设置
//使用java.util.concurrent.locks.LockSupport.getBlocker访问
volatile Object parkBlocker;
//Interruptible接口中定义了interrupt方法，用来中断指定的线程
private volatile Interruptible blocker;
//当前线程的内部锁
private final Object blockerLock = new Object();
```

```java
//线程拥有的最小优先级
public final static int MIN_PRIORITY = 1;
//线程拥有的默认优先级
public final static int NORM_PRIORITY = 5;
//线程拥有的最大优先级
public final static int MAX_PRIORITY = 10;
```

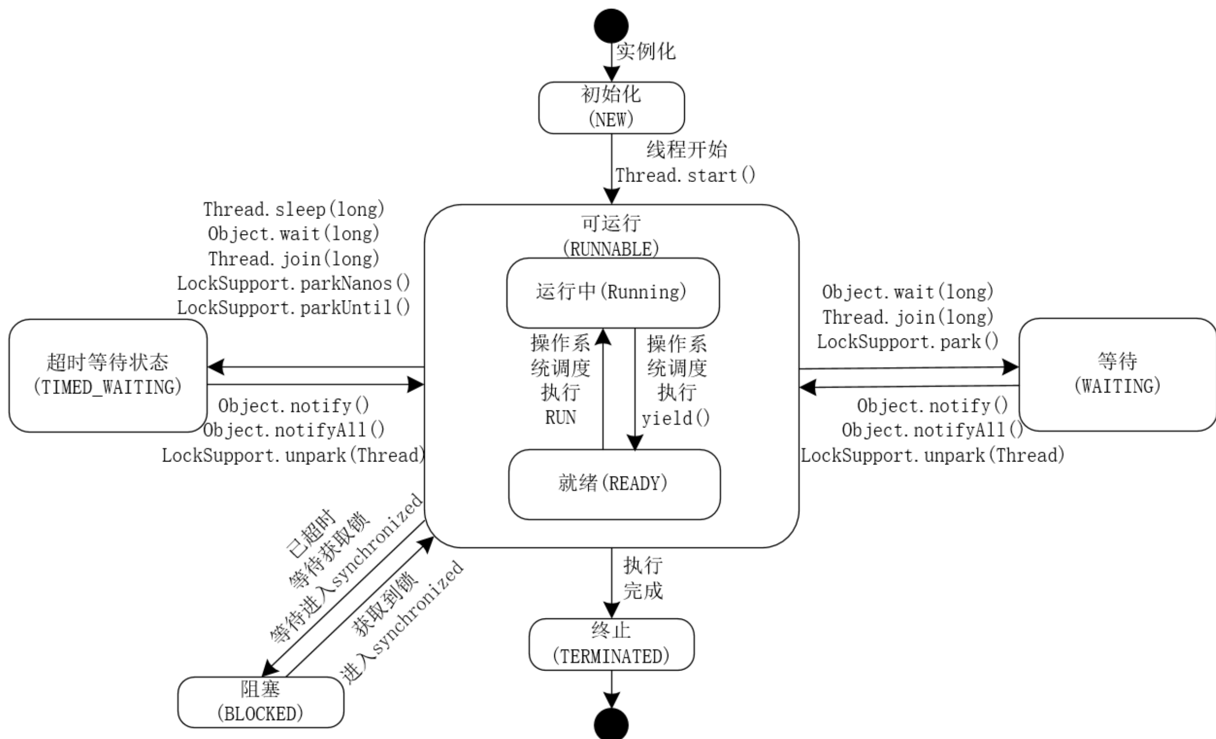
从Thread类的成员变量，我们可以看出，Thread类本质上不是一个任务，它是一个实实在在的线程对象，在Thread类中拥有一个Runnable类型的成员变量target，而这个target成员变量就是需要在Thread线程对象中执行的任务。

线程的状态定义

在Thread类的内部，定义了一个枚举State，如下所示。

```java
public enum State {
   //初始化状态
    NEW,
   //可运行状态，此时的可运行包括运行中的状态和就绪状态
    RUNNABLE,
   //线程阻塞状态
    BLOCKED,
   //等待状态
    WAITING,
    //超时等待状态
    TIMED_WAITING,
    //线程终止状态
    TERMINATED;
}
```

这个枚举类中的状态就代表了线程生命周期的各状态。我们可以使用下图来表示线程各个状态之间的转化关系。



NEW：初始状态，线程被构建，但是还没有调用start()方法。

RUNNABLE：可运行状态，可运行状态可以包括：运行中状态和就绪状态。

BLOCKED：阻塞状态，处于这个状态的线程需要等待其他线程释放锁或者等待进入synchronized。

WAITING：表示等待状态，处于该状态的线程需要等待其他线程对其进行通知或中断等操作，进而进入下一个状态。

TIME_WAITING：超时等待状态。可以在一定的时间自行返回。

TERMINATED：终止状态，当前线程执行完毕。

Thread类的构造方法

Thread类中的所有构造方法如下所示。

```java
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
Thread(Runnable target, AccessControlContext acc) {
    init(null, target, "Thread-" + nextThreadNum(), 0, acc, false);
}
public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum(), 0);
}
public Thread(String name) {
    init(null, null, name, 0);
}
public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}
public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}
public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    init(group, target, name, stackSize);
}
```

其中，我们最经常使用的就是如下几个构造方法了。

```java
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
public Thread(String name) {
    init(null, null, name, 0);
}
public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}
public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}
```

通过Thread类的源码，我们可以看出，Thread类在进行初始化的时候，都是调用的init()方法，接下来，我们看看init()方法是个啥。

init()方法

```java
private void init(ThreadGroup g, Runnable target, String name, long stackSize) {
    init(g, target, name, stackSize, null, true);
}
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals) {
    //线程的名称为空，抛出空指针异常
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }

    this.name = name;
    Thread parent = currentThread();
    //获取系统安全管理器
    SecurityManager security = System.getSecurityManager();
```

```
    //线程组为空
    if (g == null) {
        //获取的系统安全管理器不为空
        if (security != null) {
            //从系统安全管理器中获取一个线程分组
            g = security.getThreadGroup();
        }
        //线程分组为空，则从父线程获取
        if (g == null) {
            g = parent.getThreadGroup();
        }
    }
    //检查线程组的访问权限
    g.checkAccess();
    //检查权限
    if (security != null) {
        if (isCCLOverridden(getClass())) {
            security.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
        }
    }
    g.addUnstarted();
    //当前线程继承父线程的相关属性
    this.group = g;
    this.daemon = parent.isDaemon();
    this.priority = parent.getPriority();
    if (security == null || isCCLOverridden(parent.getClass()))
        this.contextClassLoader = parent.getContextClassLoader();
    else
        this.contextClassLoader = parent.contextClassLoader;
    this.inheritedAccessControlContext =
        acc != null ? acc : AccessController.getContext();
    this.target = target;
    setPriority(priority);
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
        ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    /* Stash the specified stack size in case the VM cares */
    this.stackSize = stackSize;

    //设置线程id
    tid = nextThreadID();
}
```

Thread类中的构造方法是被创建Thread线程的线程调用的，此时，调用Thread的构造方法创建线程的线程就是父线程，在init()方法中，新创建的Thread线程会继承父线程的部分属性。

run()方法

既然Thread类实现了Runnable接口，则Thread类就需要实现Runnable接口的run()方法，如下所示。

```
@Override
public void run() {
    if (target != null) {
        target.run();
    }
}
```

可以看到，Thread类中的run()方法实现非常简单，只是调用了Runnable对象的run()方法。所以，真正的任务是运行在run()方法中的。另外，**需要注意的是：直接调用Runnable接口的run()方法不会创建新线程来执行任务，如果需要创建新线程执行任务，则需要调用Thread类的start()方法。**

start()方法

```
public synchronized void start() {
   //线程不是初始化状态，则直接抛出异常
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    //添加当前启动的线程到线程组
    group.add(this);
    //标记线程是否已经启动
```

```java
    boolean started = false;
    try {
        //调用本地方法启动线程
        start0();
        //将线程是否启动标记为true
        started = true;
    } finally {
        try {
            //线程未启动成功
            if (!started) {
                //将线程在线程组里标记为启动失败
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            /* do nothing. If start0 threw a Throwable then
                  it will be passed up the call stack */
        }
    }
}

private native void start0();
```

从start()方法的源代码，我们可以看出：**start()方法使用synchronized关键字修饰，说明start()方法是同步的，它会在启动线程前检查线程的状态，如果不是初始化状态，则直接抛出异常。所以，一个线程只能启动一次，多次启动是会抛出异常的。**

这里，**也是面试的一个坑：面试官：【问题一】能不能多次调用Thread类的start()方法来启动线程吗？【问题二】多次调用Thread线程的start()方法会发生什么？【问题三】为什么会抛出异常？**

调用start()方法后，新创建的线程就会处于就绪状态（如果没有分配到CPU执行），当有空闲的CPU时，这个线程就会被分配CPU来执行，此时线程的状态为运行状态，JVM会调用线程的run()方法执行任务。

sleep()方法

sleep()方法可以使当前线程休眠，其代码如下所示。

```java
//本地方法，真正让线程休眠的方法
public static native void sleep(long millis) throws InterruptedException;

public static void sleep(long millis, int nanos)
    throws InterruptedException {
    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (nanos < 0 || nanos > 999999) {
        throw new IllegalArgumentException(
            "nanosecond timeout value out of range");
    }

    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
        millis++;
    }
    //调用本地方法
    sleep(millis);
}
```

sleep()方法会让当前线程休眠一定的时间，这个时间通常是毫秒值，这里需要注意的是：**调用sleep()方法使线程休眠后，线程不会释放相应的锁。**

join()方法

join()方法会一直等待线程超时或者终止，代码如下所示。

```java
public final synchronized void join(long millis)
    throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
```

```java
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}

public final synchronized void join(long millis, int nanos)
    throws InterruptedException {

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (nanos < 0 || nanos > 999999) {
        throw new IllegalArgumentException(
            "nanosecond timeout value out of range");
    }

    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
        millis++;
    }

    join(millis);
}

public final void join() throws InterruptedException {
    join(0);
}
```

join()方法的使用场景往往是启动线程执行任务的线程，调用执行线程的join()方法，等待执行线程执行任务，直到超时或者执行线程终止。

interrupt()方法

interrupt()方法是中断当前线程的方法，它通过设置线程的中断标志位来中断当前线程。此时，如果为线程设置了中断标志位，可能会抛出InteruptedExeption异常，同时，会清除当前线程的中断状态。这种方式中断线程比较安全，它能使正在执行的任务执行能够继续执行完毕，而不像stop()方法那样强制线程关闭。代码如下所示。

```java
public void interrupt() {
    if (this != Thread.currentThread())
        checkAccess();

    synchronized (blockerLock) {
        Interruptible b = blocker;
        if (b != null) {
            interrupt0();           // Just to set the interrupt flag
            b.interrupt(this);
            return;
        }
    }
    //调用本地方法中断线程
    interrupt0();
}
private native void interrupt0();
```

总结

作为技术人员，要知其然，更要知其所以然，我那个朋友技术本身不错，各种框架拿来就用，基本没看过常用的框架源码和JDK中常用的API，属于那种CRUD型程序员，这次面试就栽在了一个简单的Thread类上，所以，大家在学会使用的时候，一定要了解下底层的实现才好啊！

AQS中的CountDownLatch、Semaphore与CyclicBarrier

CountDownLatch

概述

同步辅助类，通过它可以阻塞当前线程。也就是说，能够实现一个线程或者多个线程一直等待，直到其他线程执行的操作完成。使用一个给定的计数器进行初始化，该计数器的操作是原子操作，即同时只能有一个线程操作该计数器。

调用该类await()方法的线程会一直阻塞，直到其他线程调用该类的countDown()方法，使当前计数器的值变为0为止。每次调用该类的countDown()方法，当前计数器的值就会减1。当计数器的值减为0的时候，所有因调用await()方法而处于等待状态的线程就会继续往下执行。这种操作只能出现一次，因为该类中的计数器不能被重置。如果需要一个可以重置计数次数的版本，可以考虑使用CyclicBarrier类。

CountDownLatch支持给定时间的等待，超过一定的时间不再等待，使用时只需要在countDown()方法中传入需要等待的时间即可。此时，countDown()方法的方法签名如下：

```
public boolean await(long timeout, TimeUnit unit)
```

使用场景

在某些业务场景中，程序执行需要等待某个条件完成后才能继续执行后续的操作。典型的应用为并行计算：当某个处理的运算量很大时，可以将该运算任务拆分成多个子任务，等待所有的子任务都完成之后，父任务再拿到所有子任务的运算结果进行汇总。

代码示例

调用ExecutorService类的shutdown()方法，并不会第一时间内把所有线程全部都销毁掉，而是让当前已有的线程全部执行完，之后，再把线程池销毁掉。

示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CountDownLatchExample {
    private static final int threadCount = 200;

    public static void main(String[] args) throws InterruptedException {

        ExecutorService exec = Executors.newCachedThreadPool();
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await();
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        Thread.sleep(100);
        log.info("{}", threadNum);
```

```
            Thread.sleep(100);
        }
    }
```

支持给定时间等待的示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
@Slf4j
public class CountDownLatchExample {
    private static final int threadCount = 200;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService exec = Executors.newCachedThreadPool();
        final CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await(10, TimeUnit.MICROSECONDS);
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        Thread.sleep(100);
        log.info("{}", threadNum);
    }
}
```

Semaphore

概述

控制同一时间并发线程的数目。能够完成对于信号量的控制，可以控制某个资源可被同时访问的个数。

提供了两个核心方法——acquire()方法和release()方法。acquire()方法表示获取一个许可，如果没有则等待，release()方法则是在操作完成后释放对应的许可。Semaphore维护了当前访问的个数，通过提供同步机制来控制同时访问的个数。Semaphore可以实现有限大小的链表。

使用场景

Semaphore常用于仅能提供有限访问的资源，比如：数据库连接数。

代码示例

每次获取并释放一个许可，示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
    private static final int threadCount = 200;
```

```
    public static void main(String[] args) throws InterruptedException {

        ExecutorService exec = Executors.newCachedThreadPool();
        final Semaphore semaphore  = new Semaphore(3);

        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    semaphore.acquire();  //获取一个许可
                    test(threadNum);
                    semaphore.release();  //释放一个许可
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        log.info("{}", threadNum);
        Thread.sleep(1000);
    }
}
```

每次获取并释放多个许可，示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
    private static final int threadCount = 200;

    public static void main(String[] args) throws InterruptedException {

        ExecutorService exec = Executors.newCachedThreadPool();
        final Semaphore semaphore  = new Semaphore(3);

        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    semaphore.acquire(3);  //获取多个许可
                    test(threadNum);
                    semaphore.release(3);  //释放多个许可
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        log.info("{}", threadNum);
        Thread.sleep(1000);
    }
}
```

假设有这样一个场景，并发太高了，即使使用Semaphore进行控制，处理起来也比较棘手。假设系统当前允许的最高并发数是3，超过3后就需要丢弃，使用Semaphore也能实现这样的场景，示例代码如下：

```java
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
    private static final int threadCount = 200;

    public static void main(String[] args) throws InterruptedException {

        ExecutorService exec = Executors.newCachedThreadPool();
        final Semaphore semaphore  = new Semaphore(3);

        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                //尝试获取一个许可，也可以尝试获取多个许可，
                    //支持尝试获取许可超时设置，超时后不再等待后续线程的执行
                    //具体可以参见Semaphore的源码
                    if (semaphore.tryAcquire()) {
                        test(threadNum);
                        semaphore.release();  //释放一个许可
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
        log.info("finish");
        exec.shutdown();
    }
    private static void test(int threadNum) throws InterruptedException {
        log.info("{}", threadNum);
        Thread.sleep(1000);
    }
}
```

CyclicBarrier

概述

是一个同步辅助类，允许一组线程相互等待，直到到达某个公共的屏障点，通过它可以完成多个线程之间相互等待，只有当每个线程都准备就绪后，才能各自继续往下执行后面的操作。

与CountDownLatch有相似的地方，都是使用计数器实现，当某个线程调用了CyclicBarrier的await()方法后，该线程就进入了等待状态，而且计数器执行加1操作，当计数器的值达到了设置的初始值，调用await()方法进入等待状态的线程会被唤醒，继续执行各自后续的操作。CyclicBarrier在释放等待线程后可以重用，所以，CyclicBarrier又被称为循环屏障。

使用场景

可以用于多线程计算数据，最后合并计算结果的场景

CyclicBarrier与CountDownLatch的区别

- CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法进行重置，并且可以循环使用
- CountDownLatch主要实现1个或n个线程需要等待其他线程完成某项操作之后，才能继续往下执行，描述的是1个或n个线程等待其他线程的关系。而CyclicBarrier主要实现了多个线程之间相互等待，直到所有的线程都满足了条件之后，才能继续执行后续的操作，描述的是各个线程内部相互等待的关系。
- CyclicBarrier能够处理更复杂的场景，如果计算发生错误，可以重置计数器让线程重新执行一次。
- CyclicBarrier中提供了很多有用的方法，比如：可以通过getNumberWaiting()方法获取阻塞的线程数量，通过isBroken()方法判断阻塞的线程是否被中断。

代码示例

示例代码如下。

```java
package io.binghe.concurrency.example.aqs;
```

```java
import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CyclicBarrierExample {

    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++){
            final int threadNum = i;
            Thread.sleep(1000);
            executorService.execute(() -> {
                try {
                    race(threadNum);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        executorService.shutdown();
    }
    private static void race(int threadNum) throws Exception{
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        cyclicBarrier.await();
        log.info("{} continue", threadNum);
    }
}
```

设置等待超时示例代码如下：

```java
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.*;
@Slf4j
public class CyclicBarrierExample {

    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++){
            final int threadNum = i;
            Thread.sleep(1000);
            executorService.execute(() -> {
                try {
                    race(threadNum);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        executorService.shutdown();
    }
    private static void race(int threadNum) throws Exception{
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        try{
            cyclicBarrier.await(2000, TimeUnit.MILLISECONDS);
        }catch (BrokenBarrierException | TimeoutException e){
            log.warn("BarrierException", e);
        }
        log.info("{} continue", threadNum);
    }
}
```

在声明CyclicBarrier的时候，还可以指定一个Runnable，当线程达到屏障的时候，可以优先执行Runnable中的方法。
示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CyclicBarrierExample {

    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5, () -> {
        log.info("callback is running");
    });

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++){
            final int threadNum = i;
            Thread.sleep(1000);
            executorService.execute(() -> {
                try {
                    race(threadNum);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        executorService.shutdown();
    }
    private static void race(int threadNum) throws Exception{
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        cyclicBarrier.await();
        log.info("{} continue", threadNum);
    }
}
```

AQS中的ReentrantLock、ReentrantReadWriteLock、StampedLock与Condition

ReentrantLock

概述

Java中主要分为两类锁，一类是synchronized修饰的锁，另外一类就是J.U.C中提供的锁。J.U.C中提供的核心锁就是ReentrantLock。

ReentrantLock（可重入锁）与synchronized区别：

（1）可重入性
二者都是同一个线程进入1次，锁的计数器就自增1，需要等到锁的计数器下降为0时，才能释放锁。

（2）锁的实现
synchronized是基于JVM实现的，而ReentrantLock是JDK实现的。

（3）性能的区别
synchronized优化之前性能比ReentrantLock差很多，但是自从synchronized引入了偏向锁，轻量级锁也就是自旋锁后，性能就差不多了。

（4）功能区别

- 便利性

synchronized使用起来比较方便，并且由编译器保证加锁和释放锁；ReentrantLock需要手工声明加锁和释放锁，最好是在finally代码块中声明释放锁。

- 锁的灵活度和细粒度

在这点上ReentrantLock会优于synchronized。

ReentrantLock独有的功能

- ReentrantLock可指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
- 提供了一个Condition类，可以分组唤醒需要唤醒的线程。而synchronized只能随机唤醒一个线程，或者唤醒全部的线程
- 提供能够中断等待锁的线程的机制，lock.lockInterruptibly()。ReentrantLock实现是一种自旋锁，通过循环调用CAS操作来实现加锁，性能上比较好是因为避免了使线程进入内核态的阻塞状态。

synchronized能做的事情ReentrantLock都能做，而ReentrantLock有些能做的事情，synchronized不能做。

在性能上，ReentrantLock不会比synchronized差。

synchronized的优势

- 不用手动释放锁，JVM自动处理，如果出现异常，JVM也会自动释放锁。
- JVM用synchronized进行管理锁定请求和释放时，JVM在生成线程转储时能够锁定信息，这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。而ReentrantLock只是普通的类，JVM不知道具体哪个线程拥有lock对象。
- synchronized可以在所有JVM版本中工作，ReentrantLock在某些1.5之前版本的JVM中可能不支持。

ReentrantLock中的部分方法说明

- boolean tryLock():仅在调用时锁定未被另一个线程保持的情况下才获取锁定。
- boolean tryLock(long, TimeUnit): 如果锁定在给定的等待时间内没有被另一个线程保持，且当前线程没有被中断，则获取这个锁定。
- void lockInterruptibly():如果当前线程没有被中断，就获取锁定；如果被中断，就抛出异常。
- boolean isLocked():查询此锁定是否由任意线程保持。
- boolean isHeldByCurrentThread(): 查询当前线程是否保持锁定状态。
- boolean isFair():判断是否是公平锁。
- boolean hasQueuedThread(Thread)：查询指定线程是否在等待获取此锁定。
- boolean hasQueuedThreads():查询是否有线程正在等待获取此锁定。
- boolean getHoldCount():查询当前线程保持锁定的个数。

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
@Slf4j
public class LockExample {
    //请求总数
    public static int clientTotal = 5000;
    //同时并发执行的线程数
    public static int threadTotal = 200;
    public static int count = 0;
    private static final Lock lock = new ReentrantLock();
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newCachedThreadPool();
        final Semaphore semaphore = new Semaphore(threadTotal);
        final CountDownLatch countDownLatch = new CountDownLatch(clientTotal);
        for(int i = 0; i < clientTotal; i++){
            executorService.execute(() -> {
                try{
                    semaphore.acquire();
                    add();
                    semaphore.release();
                }catch (Exception e){
                    log.error("exception", e);
                }
                countDownLatch.countDown();
            });
```

```
        }
        countDownLatch.await();
        executorService.shutdown();
        log.info("count:{}", count);
    }
    private static void add(){
        lock.lock();
        try{
            count ++;
        }finally {
            lock.unlock();
        }
    }
}
```

ReentrantReadWriteLock

概述

在没有任何读写锁的时候，才可以取得写锁。如果一直有读锁存在，则无法执行写锁，这就会导致写锁饥饿。

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
@Slf4j
public class LockExample {

    private final Map<String, Data> map = new TreeMap<>();
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    public Data get(String key){
        readLock.lock();
        try{
            return map.get(key);
        }finally {
            readLock.unlock();
        }
    }

    public Set<String> getAllKeys(){
        readLock.lock();
        try{
            return map.keySet();
        }finally {
            readLock.unlock();
        }
    }

    public Data put(String key, Data value){
        writeLock.lock();
        try{
            return map.put(key, value);
        }finally {
            writeLock.unlock();
        }
    }

    class Data{
```

```
    }
}
```

StampedLock

概述

控制锁三种模式：写、读、乐观读。

StampedLock的状态由版本和模式两个部分组成，锁获取方法返回的是一个数字作为票据，用相应的锁状态来表示并控制相关的访问，数字0表示没有写锁被授权访问。

在读锁上分为悲观锁和乐观锁，乐观读就是在读操作很多，写操作很少的情况下，可以乐观的认为写入和读取同时发生的几率很小。因此，不悲观的使用完全的读取锁定。程序可以查看读取资料之后，是否遭到写入进行了变更，再采取后续的措施，这样的改进可以大幅度提升程序的吞吐量。

总之，在读线程越来越多的场景下，StampedLock大幅度提升了程序的吞吐量。

StampedLock源码中的案例如下，这里加上了注释。

```
class Point {
    private double x, y;
    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) { // an exclusively locked method
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    //下面看看乐观读锁案例
    double distanceFromOrigin() { // A read-only method
        long stamp = sl.tryOptimisticRead(); //获得一个乐观读锁
        double currentX = x, currentY = y;  //将两个字段读入本地局部变量
        if (!sl.validate(stamp)) { //检查发出乐观读锁后同时是否有其他写锁发生？
            stamp = sl.readLock();  //如果没有，我们再次获得一个读悲观锁
            try {
                currentX = x; // 将两个字段读入本地局部变量
                currentY = y; // 将两个字段读入本地局部变量
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }

    //下面是悲观读锁案例
    void moveIfAtOrigin(double newX, double newY) { // upgrade
        // Could instead start with optimistic, not read mode
        long stamp = sl.readLock();
        try {
            while (x == 0.0 && y == 0.0) { //循环，检查当前状态是否符合
                long ws = sl.tryConvertToWriteLock(stamp); //将读锁转为写锁
                if (ws != 0L) { //这是确认转为写锁是否成功
                    stamp = ws; //如果成功 替换票据
                    x = newX; //进行状态改变
                    y = newY;  //进行状态改变
                    break;
                } else { //如果不能成功转换为写锁
                    sl.unlockRead(stamp);  //我们显式释放读锁
                    stamp = sl.writeLock();  //显式直接进行写锁 然后再通过循环再试
                }
            }
        } finally {
            sl.unlock(stamp); //释放读锁或写锁
        }
```

```
    }
}
```

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;
import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.StampedLock;
@Slf4j
public class LockExample {
    //请求总数
    public static int clientTotal = 5000;
    //同时并发执行的线程数
    public static int threadTotal = 200;

    public static int count = 0;

    private static final StampedLock lock = new StampedLock();

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newCachedThreadPool();
        final Semaphore semaphore = new Semaphore(threadTotal);
        final CountDownLatch countDownLatch = new CountDownLatch(clientTotal);
        for(int i = 0; i < clientTotal; i++){
            executorService.execute(() -> {
                try{
                    semaphore.acquire();
                    add();
                    semaphore.release();
                }catch (Exception e){
                    log.error("exception", e);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        log.info("count:{}", count);
    }

    private static void add(){
    //加锁时返回一个long类型的票据
        long stamp = lock.writeLock();
        try{
            count ++;
        }finally {
        //释放锁的时候带上加锁时返回的票据
            lock.unlock(stamp);
        }
    }
}
```

我们可以这样选择使用synchronozed锁还是ReentrantLock锁：

- 当只有少量竞争者时，synchronized是一个很好的通用锁实现
- 竞争者不少，但是线程的增长趋势是可预估的，此时，ReentrantLock是一个很好的通用锁实现
- synchronized不会引发死锁，其他的锁使用不当可能会引发死锁。

Condition

概述

Condition是一个多线程间协调通信的工具类，Condition除了实现wait和notify的功能以外，它的好处在于一个lock可以创建多个Condition，可以选择性的通知wait的线程

特点：

- Condition 的前提是Lock，由AQS中newCondition()方法 创建Condition的对象
- Condition await方法表示线程从AQS中移除，并释放线程获取的锁，并进入Condition等待队列中等待，等待被signal
- Condition signal方法表示唤醒对应Condition等待队列中的线程节点，并加入AQS中，准备去获取锁。

代码示例

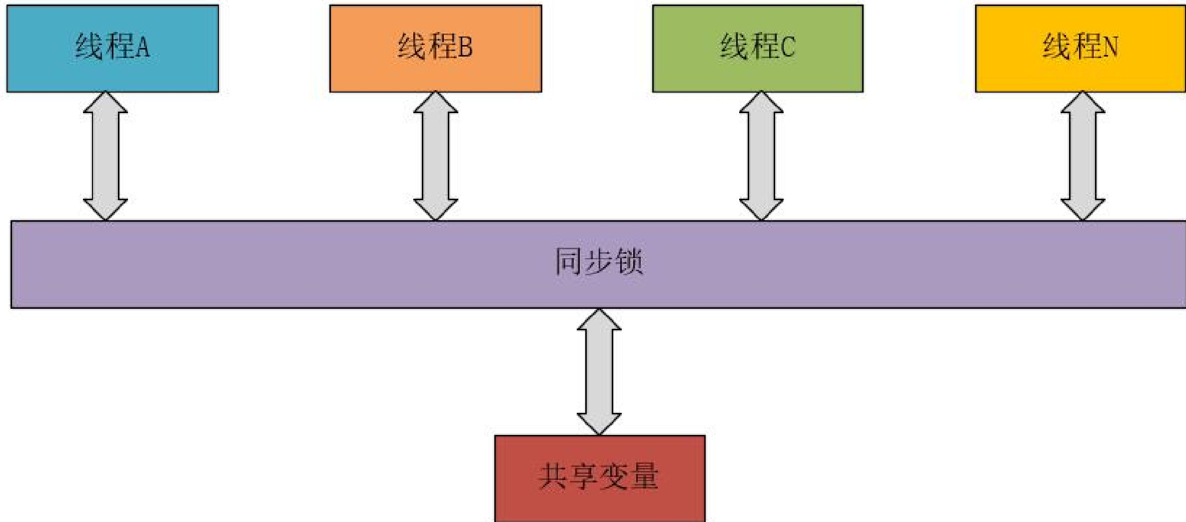
示例代码如下

```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
@Slf4j
public class LockExample {
    public static void main(String[] args) {
        ReentrantLock reentrantLock = new ReentrantLock();
        Condition condition = reentrantLock.newCondition();

        new Thread(() -> {
            try {
                reentrantLock.lock();
                log.info("wait signal"); // 1
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.info("get signal"); // 4
            reentrantLock.unlock();
        }).start();

        new Thread(() -> {
            reentrantLock.lock();
            log.info("get lock"); // 2
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            condition.signalAll();
            log.info("send signal ~ "); // 3
            reentrantLock.unlock();
        }).start();
    }
}
```

ThreadLocal学会了这些，你也能和面试官扯皮了！

前言

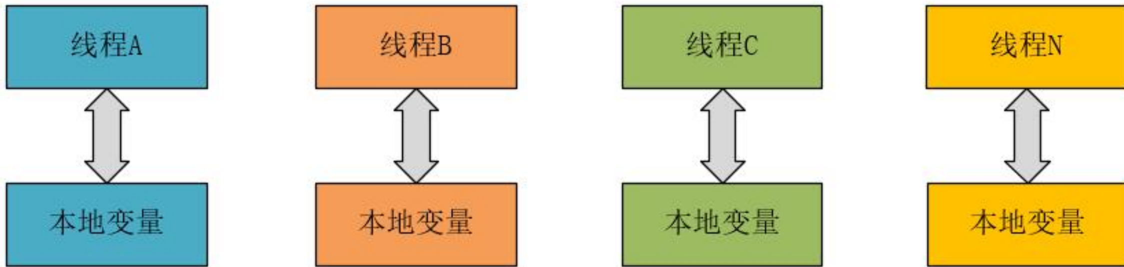
我们都知道，在多线程环境下访问同一个共享变量，可能会出现线程安全的问题，为了保证线程安全，我们往往会在访问这个共享变量的时候加锁，以达到同步的效果，如下图所示。



对共享变量加锁虽然能够保证线程的安全，但是却增加了开发人员对锁的使用技能，如果锁使用不当，则会导致死锁的问题。而**ThreadLocal能够做到在创建变量后，每个线程对变量访问时访问的是线程自己的本地变量。**

什么是ThreadLocal?

> ThreadLocal是JDK提供的，支持线程本地变量。也就是说，如果我们创建了一个ThreadLocal变量，则访问这个变量的每个线程都会有这个变量的一个本地副本。如果多个线程同时对这个变量进行读写操作时，实际上操作的是线程自己本地内存中的变量，从而避免了线程安全的问题。



ThreadLocal使用示例

例如，我们使用ThreadLocal保存并打印相关的变量信息，程序如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA: " + Thread.currentThread().getName());
            System.out.println("线程A本地变量中的值为: " + threadLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB: " + Thread.currentThread().getName());
            System.out.println("线程B本地变量中的值为: " + threadLocal.get());
        });
        //启动线程A和线程B
        threadA.start();
        threadB.start();
    }
}
```

运行程序，打印的结果信息如下所示。

```
线程A本地变量中的值为：ThreadA：Thread-0
线程B本地变量中的值为：ThreadB：Thread-1
```

此时，我们为线程A增加删除ThreadLocal中的变量的操作，如下所示。

```java
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA：" + Thread.currentThread().getName());
            System.out.println("线程A本地变量中的值为：" + threadLocal.get());
            threadLocal.remove();
            System.out.println("线程A删除本地变量后ThreadLocal中的值为：" + threadLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB：" + Thread.currentThread().getName());
            System.out.println("线程B本地变量中的值为：" + threadLocal.get());
            System.out.println("线程B没有删除本地变量：" + threadLocal.get());
        });
        //启动线程A和线程B
        threadA.start();
        threadB.start();
    }
}
```

此时的运行结果如下所示。

```
线程A本地变量中的值为：ThreadA：Thread-0
线程B本地变量中的值为：ThreadB：Thread-1
线程B没有删除本地变量：ThreadB：Thread-1
线程A删除本地变量后ThreadLocal中的值为：null
```

通过上述程序我们可以看出，**线程A和线程B存储在ThreadLocal中的变量互不干扰，线程A存储的变量只能由线程A访问，线程B存储的变量只能由线程B访问。**

ThreadLocal原理

首先，我们看下Thread类的源码，如下所示。

```java
public class Thread implements Runnable {
    /***********省略N行代码*************/
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
     /***********省略N行代码*************/
}
```

由Thread类的源码可以看出，在ThreadLocal类中存在成员变量threadLocals和inheritableThreadLocals，这两个成员变量都是ThreadLocalMap类型的变量，而且二者的初始值都为null。只有当前线程第一次调用ThreadLocal的set()方法或者get()方法时才会实例化变量。

这里需要注意的是：**每个线程的本地变量不是存放在ThreadLocal实例里面的，而是存放在调用线程的threadLocals变量里面的。** 也就是说，调用ThreadLocal的set()方法存储的本地变量是存放在具体线程的内存空间中的，而ThreadLocal类只是提供了set()和get()方法来存储和读取本地变量的值，当调用ThreadLocal类的set()方法时，把要存储的值放入**调用线程**的threadLocals中存储起来，当调用ThreadLocal类的get()方法时，从当前线程的threadLocals变量中将存储的值取出来。

接下来，我们分析下ThreadLocal类的set()、get()和remove()方法的实现逻辑。

set()方法

set()方法的源代码如下所示。

```java
public void set(T value) {
    //获取当前线程
    Thread t = Thread.currentThread();
    //以当前线程为Key，获取ThreadLocalMap对象
    ThreadLocalMap map = getMap(t);
    //获取的ThreadLocalMap对象不为空
    if (map != null)
        //设置value的值
        map.set(this, value);
    else
        //获取的ThreadLocalMap对象为空，创建Thread类中的threadLocals变量
        createMap(t, value);
}
```

在set()方法中，首先获取调用set()方法的线程，接下来，使用当前线程作为Key调用getMap(t)方法来获取ThreadLocalMap对象，getMap(Thread t)的方法源码如下所示。

```java
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

可以看到，getMap(Thread t)方法获取的是线程变量自身的threadLocals成员变量。

在set()方法中，如果调用getMap(t)方法返回的对象不为空，则把value值设置到Thread类的threadLocals成员变量中，而传递的key为当前ThreadLocal的this对象，value就是通过set()方法传递的值。

如果调用getMap(t)方法返回的对象为空，则程序调用createMap(t, value)方法来实例化Thread类的threadLocals成员变量。

```java
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

也就是创建当前线程的threadLocals变量。

get()方法

get()方法的源代码如下所示。

```java
public T get() {
    //获取当前线程
    Thread t = Thread.currentThread();
    //获取当前线程的threadLocals成员变量
    ThreadLocalMap map = getMap(t);
    //获取的threadLocals变量不为空
    if (map != null) {
        //返回本地变量对应的值
        ThreadLocalMap.Entry e = map.getEntry(this);
```

```
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //初始化threadLocals成员变量的值
    return setInitialValue();
}
```

通过当前线程来获取threadLocals成员变量，如果threadLocals成员变量不为空，则直接返回当前线程绑定的本地变量，否则调用setInitialValue()方法初始化threadLocals成员变量的值。

```
private T setInitialValue() {
    //调用初始化Value的方法
    T value = initialValue();
    Thread t = Thread.currentThread();
    //根据当前线程获取threadLocals成员变量
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //threadLocals不为空，则设置value值
        map.set(this, value);
    else
        //threadLocals为空,创建threadLocals变量
        createMap(t, value);
    return value;
}
```

其中，initialValue()方法的源码如下所示。

```
protected T initialValue() {
    return null;
}
```

通过initialValue()方法的源码可以看出，这个方法可以由子类覆写，在ThreadLocal类中，这个方法直接返回null。

remove()方法

remove()方法的源代码如下所示。

```
public void remove() {
    //根据当前线程获取threadLocals成员变量
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        //threadLocals成员变量不为空，则移除value值
        m.remove(this);
}
```

remove()方法的实现比较简单，首先根据当前线程获取threadLocals成员变量，不为空，则直接移除value的值。

注意：如果调用线程一致不终止，则本地变量会一直存放在调用线程的threadLocals成员变量中，所以，如果不需要使用本地变量时，可以通过调用ThreadLocal的remove()方法，将本地变量从当前线程的threadLocals成员变量中删除，以免出现内存溢出的问题。

ThreadLocal变量不具有传递性

> 使用ThreadLocal存储本地变量不具有传递性，也就是说，同一个ThreadLocal在父线程中设置值后，在子线程中是无法获取到这个值的，这个现象说明ThreadLocal中存储的本地变量不具有传递性。

接下来，我们来看一段代码，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //在主线程中设置值
        threadLocal.set("ThreadLocalTest");
        //在子线程中获取值
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("子线程获取值：" + threadLocal.get());
            }
        });
        //启动子线程
        thread.start();
        //在主线程中获取值
        System.out.println("主线程获取值：" + threadLocal.get());
    }
}
```

运行这段代码输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：null
```

通过上述程序，我们可以看出在主线程中向ThreadLocal设置值后，在子线程中是无法获取到这个值的。**那有没有办法在子线程中获取到主线程设置的值呢？此时，我们可以使用InheritableThreadLocal来解决这个问题。**

InheritableThreadLocal使用示例

InheritableThreadLocal类继承自ThreadLocal类，它能够让子线程访问到在父线程中设置的本地变量的值，例如，我们将ThreadLocalTest类中的threadLocal静态变量改写成InheritableThreadLocal类的实例，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new InheritableThreadLocal<String>();

    public static void main(String[] args){
        //在主线程中设置值
        threadLocal.set("ThreadLocalTest");
        //在子线程中获取值
        Thread thread = new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                System.out.println("子线程获取值：" + threadLocal.get());
            }
        });
        //启动子线程
        thread.start();
        //在主线程中获取值
        System.out.println("主线程获取值：" + threadLocal.get());
    }
}
```

此时，运行程序输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：ThreadLocalTest
```

可以看到，**使用InheritableThreadLocal类存储本地变量时，子线程能够获取到父线程中设置的本地变量。**

双击评论 666

InheritableThreadLocal原理

首先，我们来看下InheritableThreadLocal类的源码，如下所示。

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
    protected T childValue(T parentValue) {
        return parentValue;
    }

    ThreadLocalMap getMap(Thread t) {
       return t.inheritableThreadLocals;
    }

    void createMap(Thread t, T firstValue) {
        t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);
    }
}
```

由InheritableThreadLocal类的源代码可知，InheritableThreadLocal类继承自ThreadLocal类，并且重写了ThreadLocal类的childValue()方法、getMap()方法和createMap()方法。也就是说，当调用ThreadLocal的set()方法时，创建的是当前Thread线程的inheritableThreadLocals成员变量而不再是threadLocals成员变量。

这里，**我们需要思考一个问题：InheritableThreadLocal类的childValue()方法是何时被调用的呢？**这就需要我们来看下Thread类的构造方法了，如下所示。

```
public Thread() {
     init(null, null, "Thread-" + nextThreadNum(), 0);
 }

public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

```java
Thread(Runnable target, AccessControlContext acc) {
    init(null, target, "Thread-" + nextThreadNum(), 0, acc, false);
}

public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum(), 0);
}

public Thread(String name) {
    init(null, null, name, 0);
}

public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}

public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    init(group, target, name, stackSize);
}
```

可以看到，Thread类的构造方法最终调用的是init()方法，那我们就来看下init()方法，如下所示。

```java
private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc,
                      boolean inheritThreadLocals) {
       /************省略部分源码************/
        if (inheritThreadLocals && parent.inheritableThreadLocals != null)
            this.inheritableThreadLocals =
                ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
        /* Stash the specified stack size in case the VM cares */
        this.stackSize = stackSize;

        /* Set thread ID */
        tid = nextThreadID();
    }
```

可以看到，在init()方法中会判断传递的inheritThreadLocals变量是否为true，同时父线程中的inheritableThreadLocals是否为null，如果传递的inheritThreadLocals变量为true，同时，父线程中的inheritableThreadLocals不为null，则调用ThreadLocal类的createInheritedMap()方法。

```java
static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

在createInheritedMap()中，使用父线程的inheritableThreadLocals变量作为参数创建新的ThreadLocalMap对象。然后在Thread类的init()方法中会将这个ThreadLocalMap对象赋值给子线程的inheritableThreadLocals成员变量。

接下来，我们来看看ThreadLocalMap的构造函数都干了啥，如下所示。

```java
private ThreadLocalMap(ThreadLocalMap parentMap) {
    Entry[] parentTable = parentMap.table;
    int len = parentTable.length;
    setThreshold(len);
    table = new Entry[len];

    for (int j = 0; j < len; j++) {
        Entry e = parentTable[j];
        if (e != null) {
            @SuppressWarnings("unchecked")
            ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
```

```
            if (key != null) {
                //调用重写的childValue方法
                Object value = key.childValue(e.value);
                Entry c = new Entry(key, value);
                int h = key.threadLocalHashCode & (len - 1);
                while (table[h] != null)
                    h = nextIndex(h, len);
                table[h] = c;
                size++;
            }
        }
    }
}
```

在ThreadLocalMap的构造函数中，调用了InheritableThreadLocal类重写的childValue()方法。而InheritableThreadLocal类通过重写getMap()方法和createMap()方法，让本地变量保存到了Thread线程的inheritableThreadLocals变量中，线程通过InheritableThreadLocal类的set()方法和get()方法设置变量时，就会创建当前线程的inheritableThreadLocals变量。此时，如果父线程创建子线程，在Thread类的构造函数中会把父线程中的inheritableThreadLocals变量里面的本地变量复制一份保存到子线程的inheritableThreadLocals变量中。

又一个朋友面试栽在了Thread类的stop()方法和interrupt()方法上！

一个工作了几年的朋友今天打电话和我聊天，说前段时间出去面试，面试官问他做过的项目，他讲起业务来那是头头是道，犹如滔滔江水连绵不绝，可面试官最后问了一个问题：Thread类的stop()方法和interrupt方法有啥区别。这一问不要紧，当场把那个朋友打懵了！结果可想而知。。。

事后，我也是感慨颇多，现在的程序员只知道做些简单的CRUD吗？哎，不多说了，今天就简单的说说Thread类的stop()方法和interrupt()方法到底有啥区别吧！

stop()方法

stop()方法会真的杀死线程。如果线程持有ReentrantLock锁，被stop()的线程并不会自动调用ReentrantLock的unlock()去释放锁，那其他线程就再也没机会获得ReentrantLock锁， 这样其他线程就再也不能执行ReentrantLock锁锁住的代码逻辑。 所以该方法就不建议使用了， 类似的方法还有suspend()和resume()方法， 这两个方法同样也都不建议使用了， 所以这里也就不多介绍了。

interrupt()方法

interrupt()方法仅仅是通知线程，线程有机会执行一些后续操作，同时也可以无视这个通知。被interrupt的线程，有两种方式接收通知：**一种是异常， 另一种是主动检测。**

通过异常接收通知

当线程A处于WAITING、 TIMED_WAITING状态时， 如果其他线程调用线程A的interrupt()方法，则会使线程A返回到RUNNABLE状态，同时线程A的代码会触发InterruptedException异常。线程转换到WAITING、TIMED_WAITING状态的触发条件，都是调用了类似wait()、join()、sleep()这样的方法， 我们看这些方法的签名时，发现都会throws InterruptedException这个异常。这个异常的触发条件就是：其他线程调用了该线程的interrupt()方法。

当线程A处于RUNNABLE状态时，并且阻塞在java.nio.channels.InterruptibleChannel上时， 如果其他线程调用线程A的interrupt()方法，线程A会触发java.nio.channels.ClosedByInterruptException这个异常；当阻塞在java.nio.channels.Selector上时，如果其他线程调用线程A的interrupt()方法，线程A的java.nio.channels.Selector会立即返回。

主动检测通知

如果线程处于RUNNABLE状态，并且没有阻塞在某个I/O操作上，例如中断计算基因组序列的线程A，此时就得依赖线程A主动检测中断状态了。如果其他线程调用线程A的interrupt()方法， 那么线程A可以通过isInterrupted()方法， 来检测自己是不是被中断了。

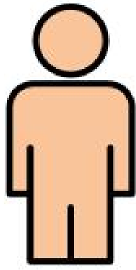
基础案例篇

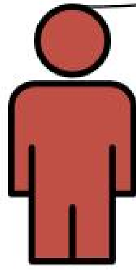
工作了3年的程序员小菜面试高并发岗位被吊打虐哭

写在前面

程序员小菜是一家互联网公司的开发人员，主要负责后端Java技术开发，平时的工作中以CRUD为主。从刚毕业来到公司，一转眼3年过去了，小菜突然觉得在这家公司工作没啥意思了，整天做CRUD的工作没啥挑战。于是，小菜童鞋优化了下自己的简历，并在网上投递了自己的简历，不一会，一个电话打过来，对方传来一个软萌妹纸的声音。

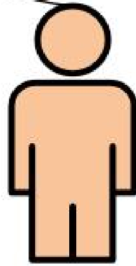
您好，请问是菜XX吗？

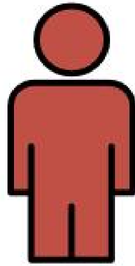




小菜　　HR

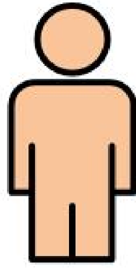
您好，我是，请问您是？

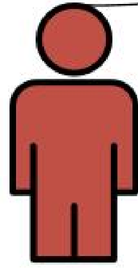




小菜　　HR

菜先生，您好，我是XX公司的HR，您的简历通过了我们的筛选，请问您明天上午有时间来我们公司面试吗？

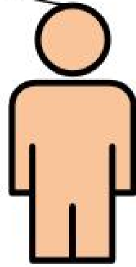


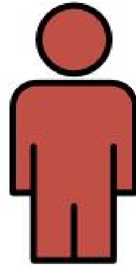


小菜　　　　HR

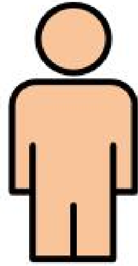
小菜心中一阵狂喜！！！

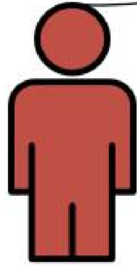
有时间，有时间

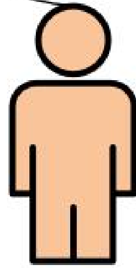


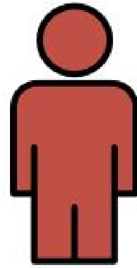


小菜　　　　HR



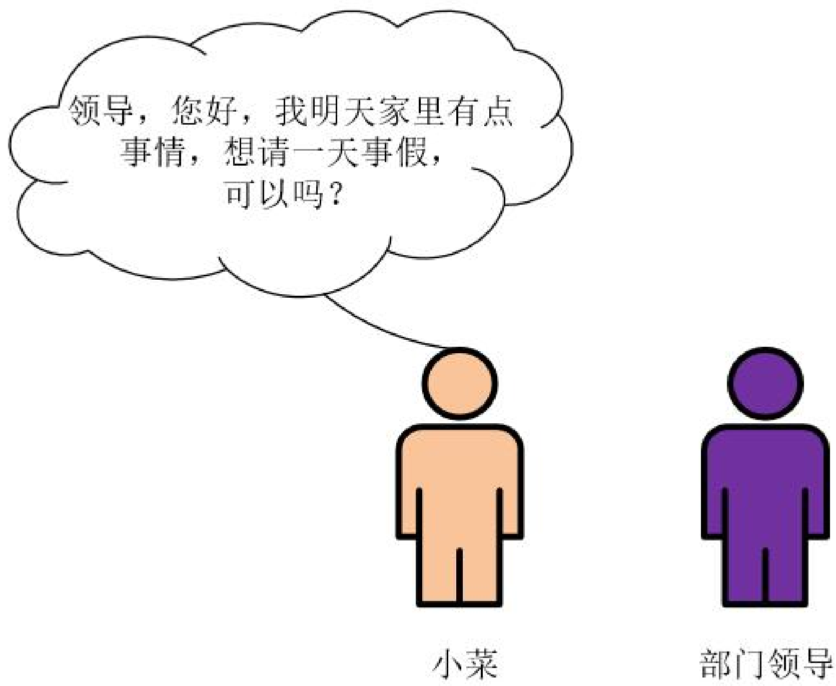


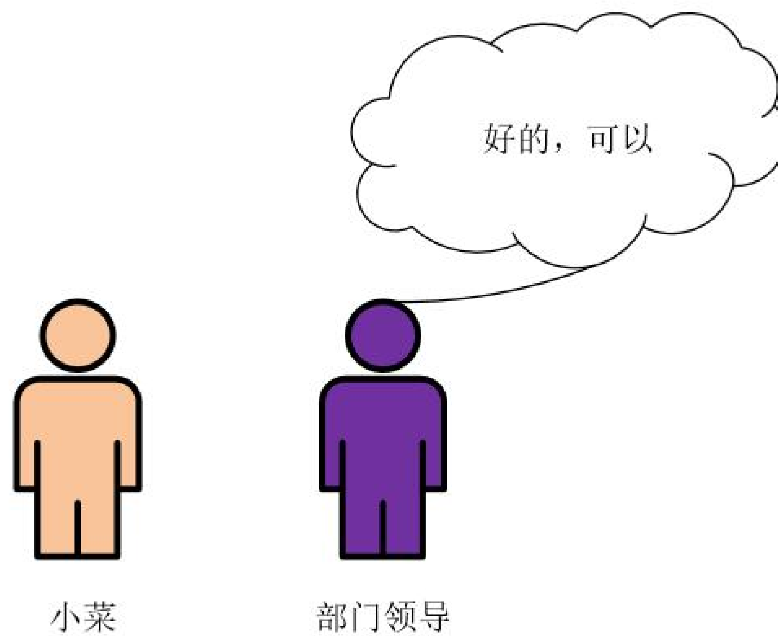




。。。嘟嘟嘟，电话已挂掉。。。

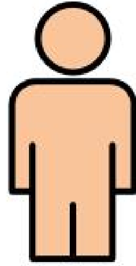
小菜童鞋心中一阵狂喜，并果断向部门领导请了一天假，信心满满的去面试了。

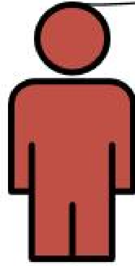


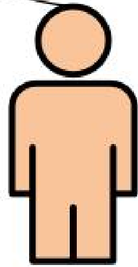


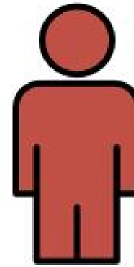
面试经过

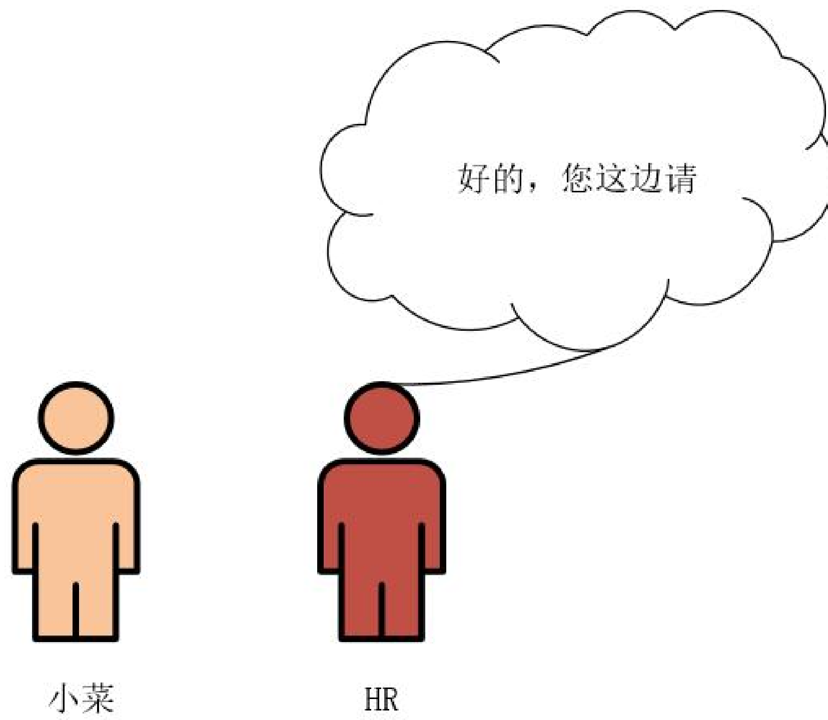
小菜按时来到了XX公司，迎接他的正式昨天打电话通知他来面试的HR妹纸。


小菜

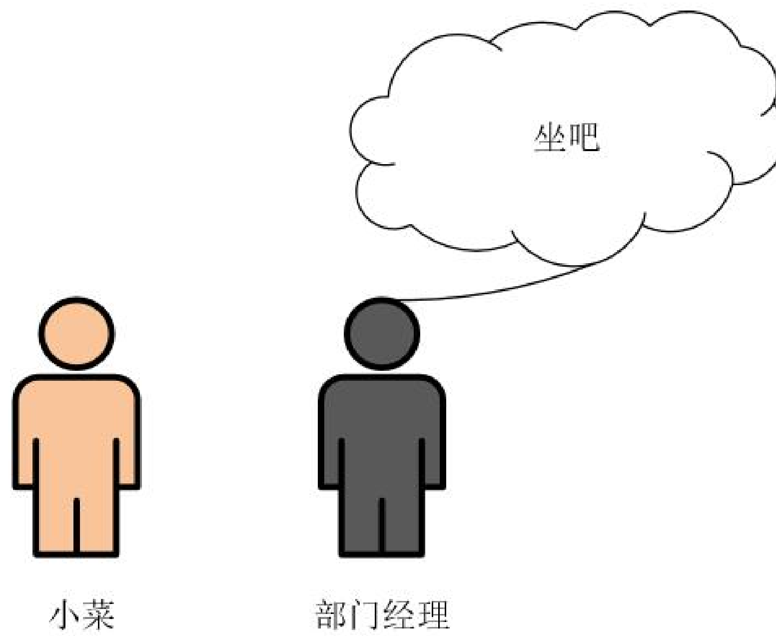

您好，请问您是菜先生吧？
HR


您好，我是菜XX，我是来面试Java开发岗位的
小菜


HR

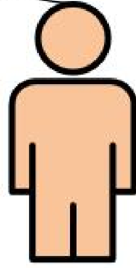


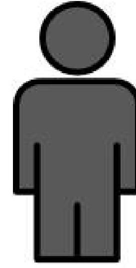
小菜被带进了部门经理的办公室，一进门，小菜就看见办公桌对面坐着一个"地中海式"发型的大牛，一看就是大神级别的啊！头发都没了，脑门油光油光的！部门经理冲小菜笑了笑。



面对这样的大神，此时的小菜心里还是有点紧张的，小菜在部门经理的对面坐了下来，并双手递上了自己的简历。

面试官，您好，我是来
面试Java开发岗位的

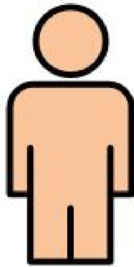


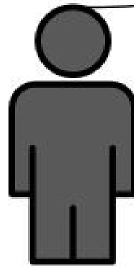


小菜　　部门经理

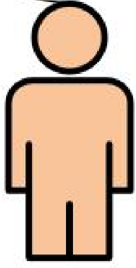
部门经理结果简历，边看边说。

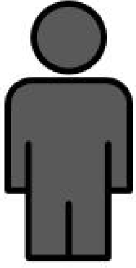
先做个自我介绍吧

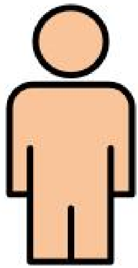


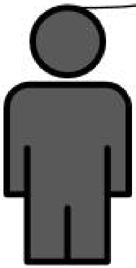


小菜　　部门经理

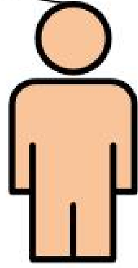

好的，我叫菜xx，3年工作经验，熟悉Java开发、熟练掌握了Spring、SpringBoot，吧啦吧啦。。。
小菜


部门经理


小菜

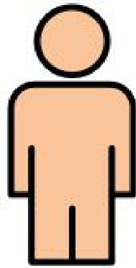

对高并发这块有做过相应的研究吗
部门经理

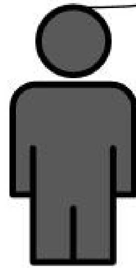
个人感觉这块还可以吧



小菜　　部门经理

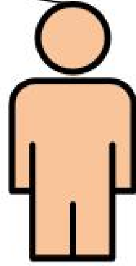
那好，你说一下什么是线程？什么是进程吧？



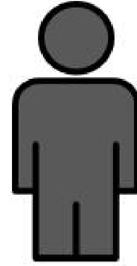


小菜　　部门经理

进程就是。。。
线程就是。。。
它们二者的关系是。。。
吧啦吧啦。。。

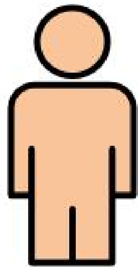


小菜

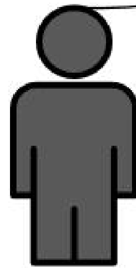


部门经理

那你说一下进程间是如何通讯的呢？



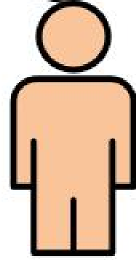
小菜

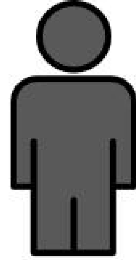


部门经理

小菜一听，完了，这个根本就不知道啊。。。

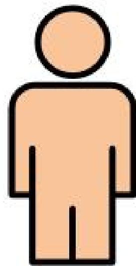
额，这个确实没深入研究过。。。

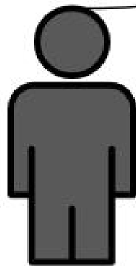




小菜　　部门经理

没关系，那你知道 ConcurrentHashMap 和 HashTable有什么区别吗？

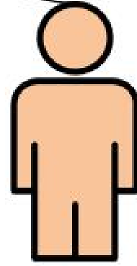




小菜　　部门经理

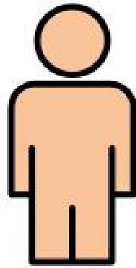
小菜心里有点慌。。。

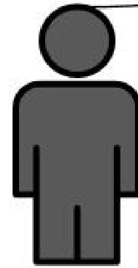
额，这个也没深入研究过。。。



小菜　部门经理

没关系，不用紧张，你说说如何在两个线程间共享数据？

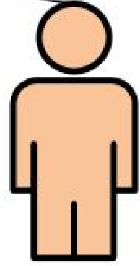


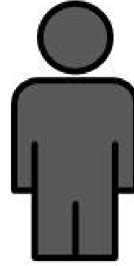


小菜　部门经理

小菜感觉自己脑门开始冒汗了。。。

额，这个也没深入研究过。。。

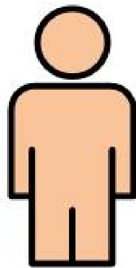


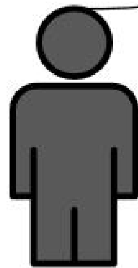


小菜

部门经理

为什么wait和notify方法要在同步块代码中调用？

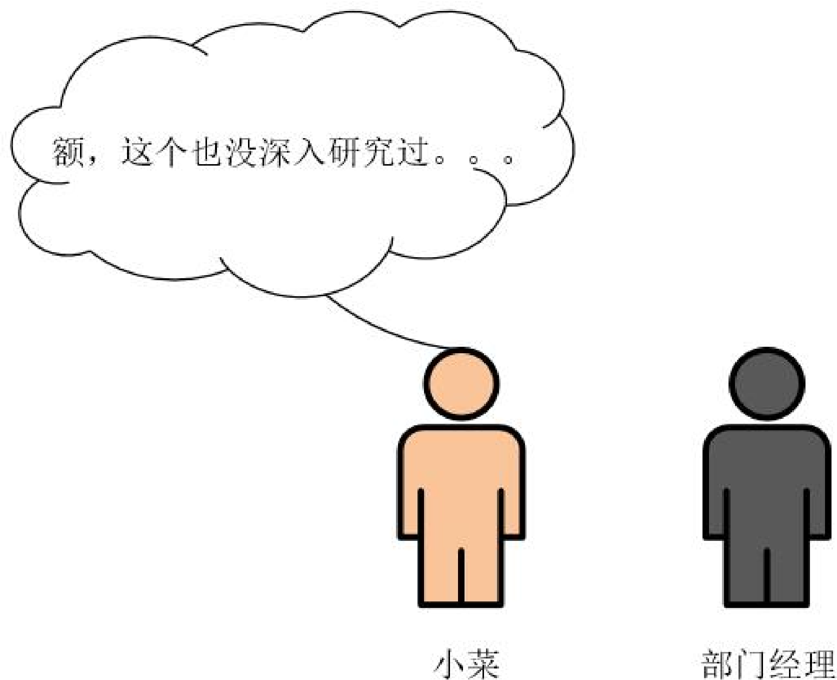




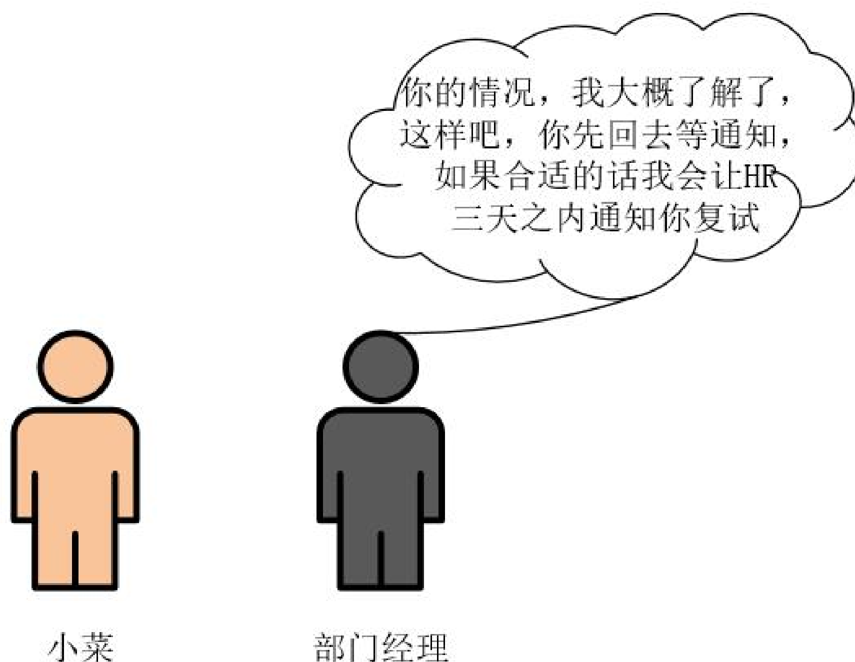
小菜

部门经理

小菜俨然一脸懵逼。。。



此时，部门经理抬起头，笑着对小菜说。



小菜一听就知道这次面试肯定是黄了，问的问题自己啥都不会，心里很不是个滋味！简直是欲哭无泪啊！小菜从座位上起来，走出办公室，径直走出了这家公司。在回去的路上，小菜心情沉重，想想十几分钟之前，还信心满满的过来面试，没想到Java并发编程会涉及到这么多的知识，自己在工作中完全没接触到啊！

此时，小菜突然想起来自己的好朋友——大冰，一个工作经验丰富的大神级别的人物，小菜决定要跟他学习高并发编程的知识！努力提升自己的编程技能！从此，小菜踏上了高并发学习的道路。

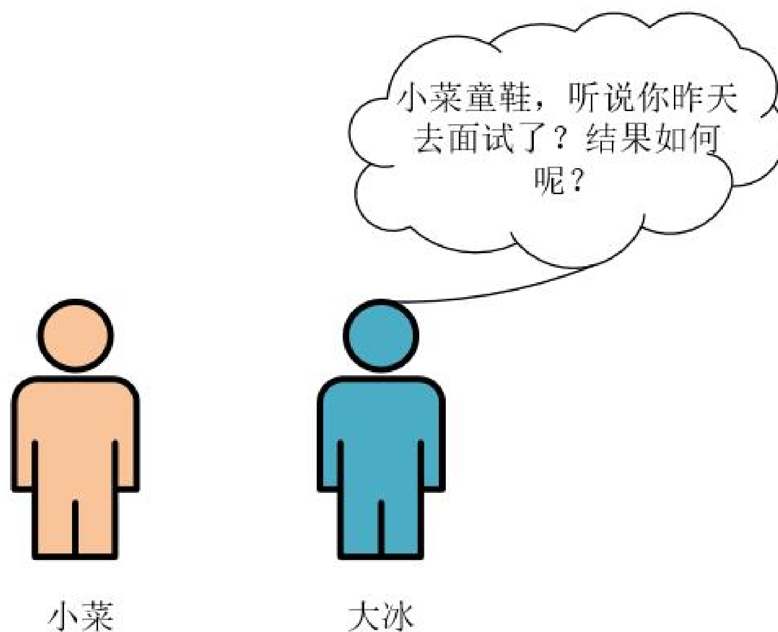
总结

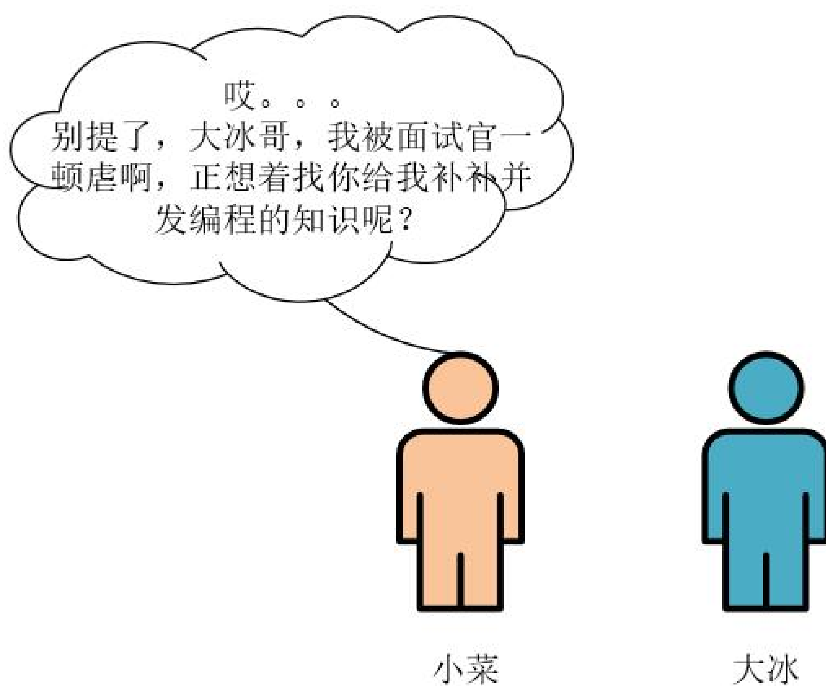
并发编程一直是让人很头疼的事情，很多人总觉得自己似乎掌握了并发编程的知识，就像文中的小菜那样，信心满满的去面试，却被面试官吊打虐哭。所以，并发编程需要我们静下心来，认真研读每一个知识点，将每个知识点研究透彻，由点到线，再由线连成面，形成自己的知识体系。深入掌握并发编程的技能之后，到时候，就是你吊打面试官了！

导致并发编程频繁出问题的“幕后黑手”

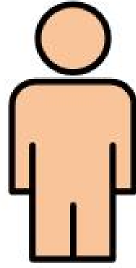
写在前面

> 工作了3年的小菜同学，平时在公司只是做些CRUD的常规工作，这次，出去面试被面试官一顿虐啊！尤其是并发编程的知识简直就是被吊打啊。小菜心有不甘，回来找自己工作经验非常丰富的朋友大冰来帮助自己提升并发编程的知识，于是便有了接下来的一系列小菜学并发的文章。

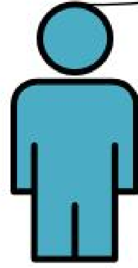




现在并发知识确实是Java开发中的必备技能呀，但是学习并发编程很辛苦，你能坚持吗？

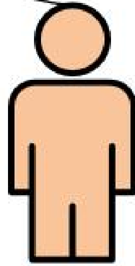


小菜

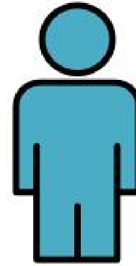


大冰

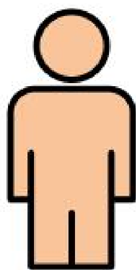
哎。。。
大冰哥，通过这次面试我觉得自己太菜了，我一定要坚持学并发知识，提高自己的知识技能

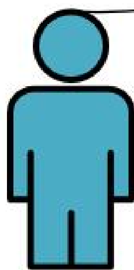


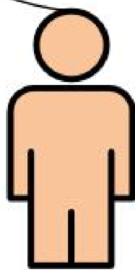
小菜

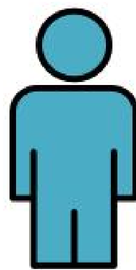


大冰









并发编程的难点

并发编程一直是很让人头疼的问题，因为多线程环境下不太好定位问题，它不像一般的业务代码那样打个断点，debug一下基本就能够定位问题所在。并发编程中，出现的问题往往都是很诡异的，而且大多数情况下，问题也不是每次都会重现的。那么，我们如何才能够更好的解决并发问题呢？这就需要我们了解造成这些问题的"幕后黑手"究竟是什么！

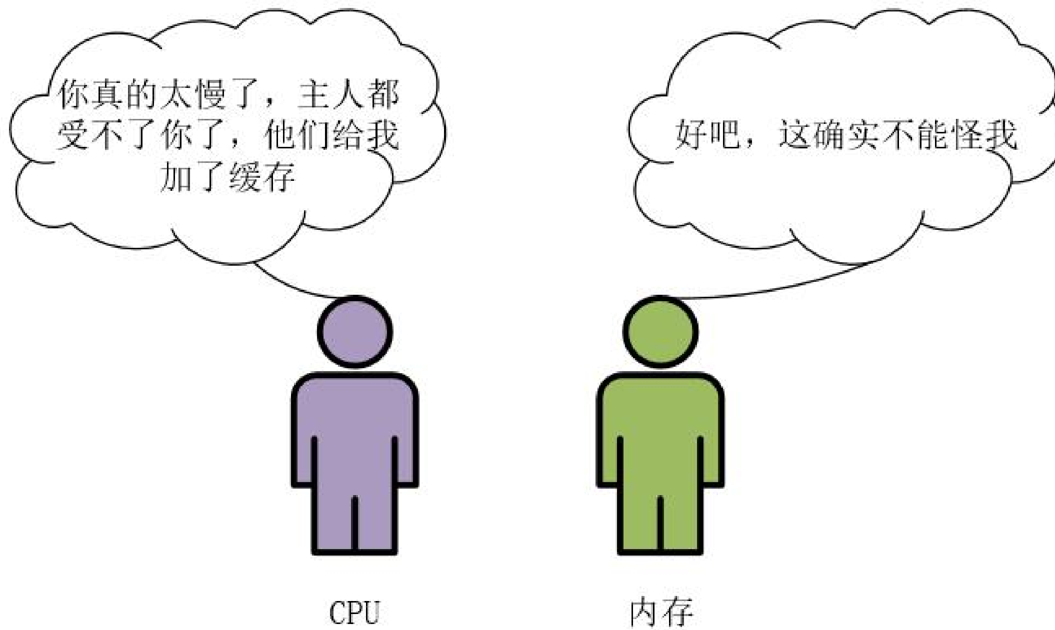
操作系统做出的努力

计算机的制造商为了提升计算机的整体性能，对计算机做出了相应的优化措施。

CPU增加了缓存

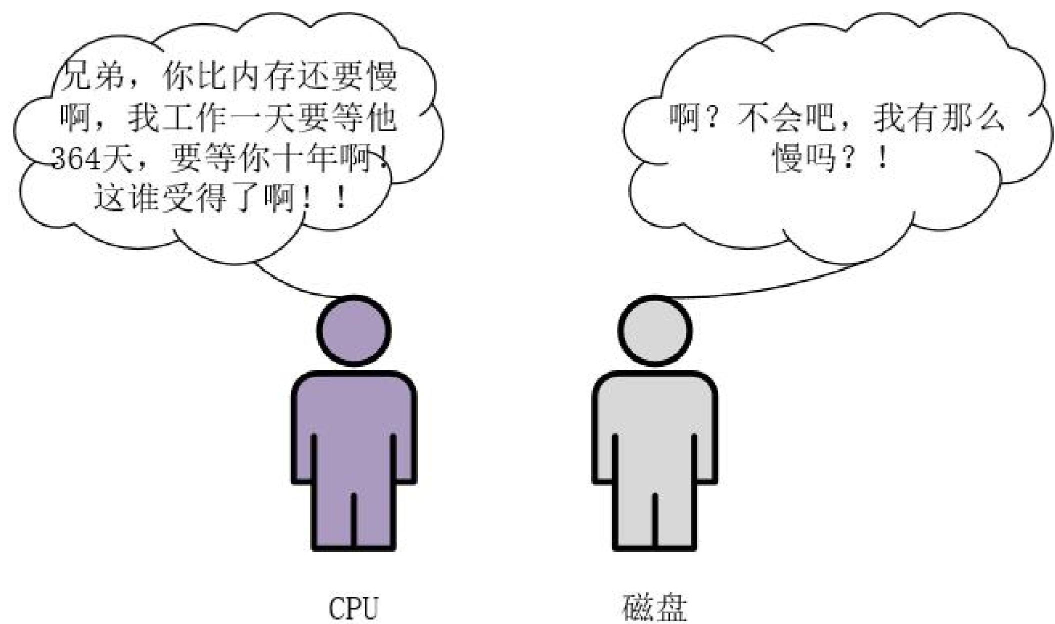
CPU对于数据的计算速度远远高于从内存中存取数据的速度，我们可以想象一下，如果说，CPU对于数据的计算需要一天时间，那么，从内存中读取和写入数据可能分别需要1年的时间。也就是说，在这一年365天当中，CPU只需要工作一天，而364天的时间都在等待从内存中读取或者存储数据。这对于CPU的利用率来说，就是极大的浪费了。

为了缓和CPU与内存之间的速度差异，计算机的制造商为CPU增加了缓存。

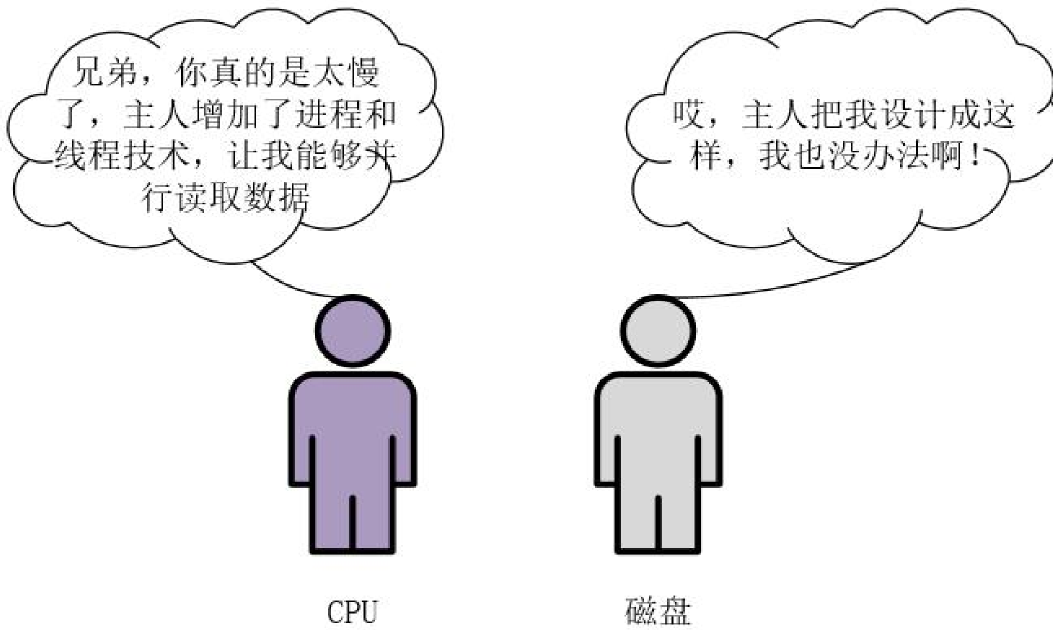


操作系统增加了进程和线程

CPU的速度比内存快的多，而内存又比磁盘快的多。如果说，CPU运算需要一天的时间，从内存读取数据需要一年的话，那从磁盘读取数据就需要10年的时间了。没错，磁盘就是这么慢啊！

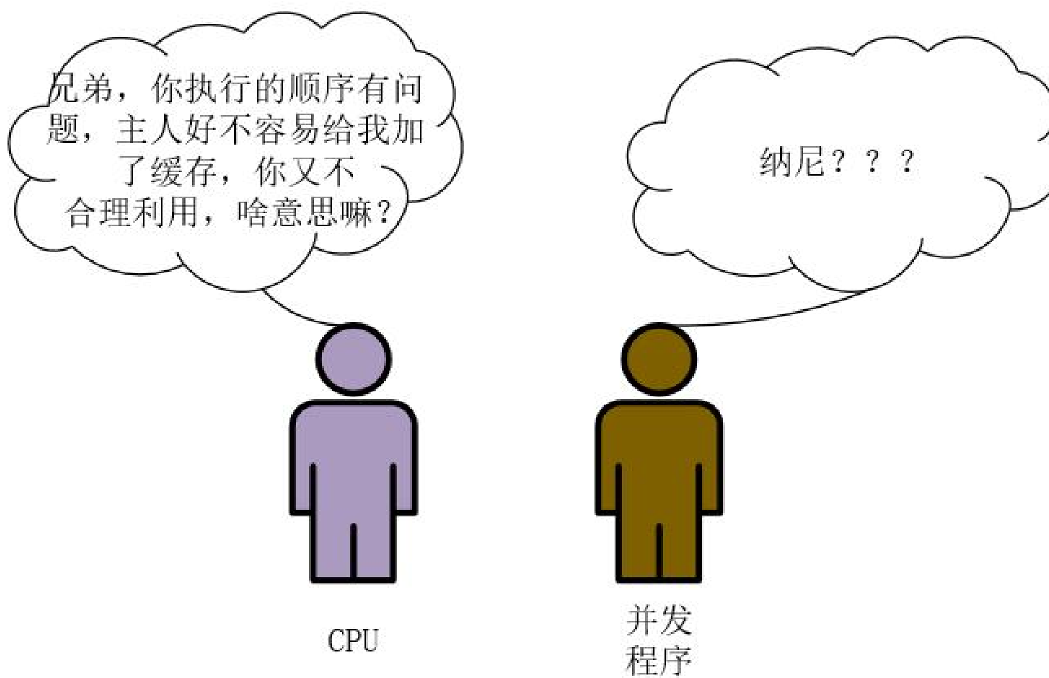


为了缓和CPU和磁盘设备之间的速度差异，操作系统的制造商增加了进程和线程技术。

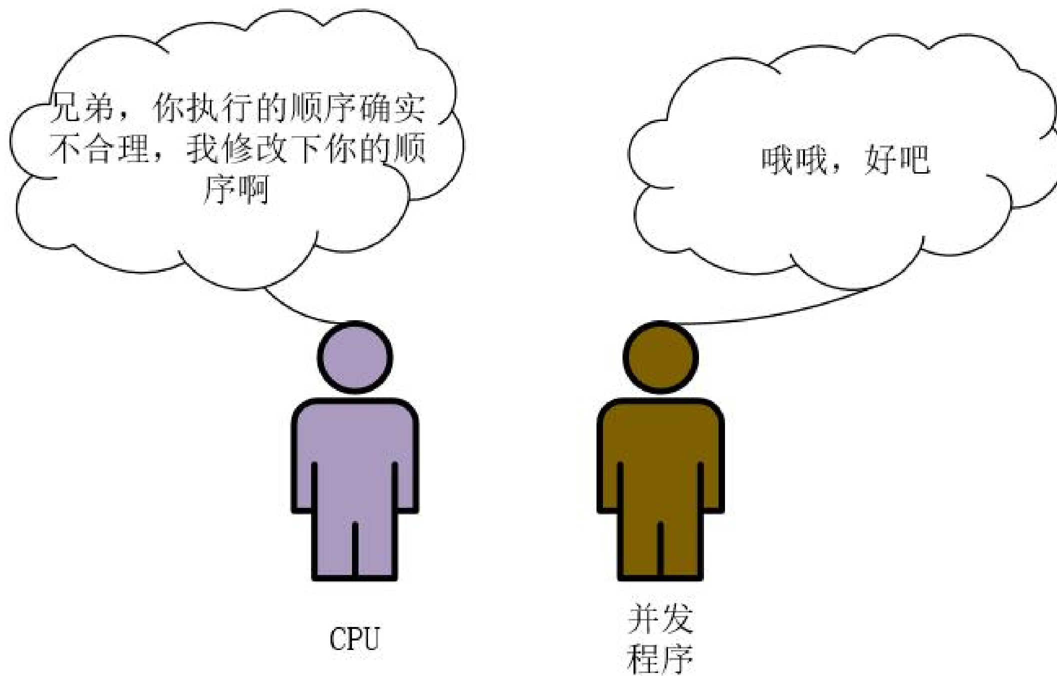


优化CPU指令执行顺序

我们写的并发程序在操作系统上运行时，对于CPU缓存的使用可能会不太合理，造成CPU缓存的浪费。

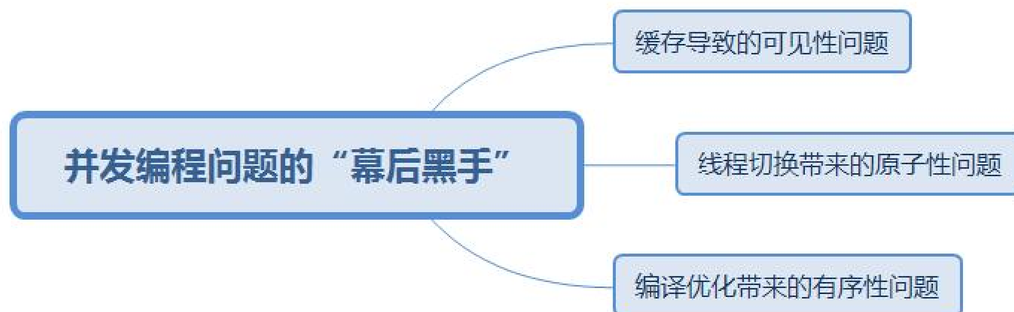


为了使CPU的缓存能够得到更加合理的利用，编译程序对CPU上指令的执行顺序进行了优化。



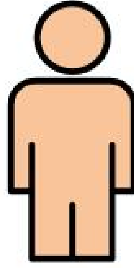
揪出“幕后”黑手

计算机和操作系统的制造商对计算机和操作系统进行的优化，在无形当中造成了很多并发编程的问题。本质上，并发编程的很多诡异的问题源头也在于此，这也是并发编程频繁出问题的“幕后黑手”。所以，我们可以将“幕后黑手”总结如下。

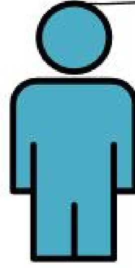


没错，这就是造成并发编程问题的“幕后黑手”！！后面我们将分别细化说明这些“幕后黑手”。

好了，今天就讲到这吧！你回去好好消化消化，后面我们再继续。

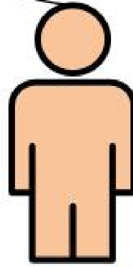


小菜

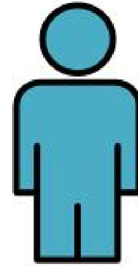


大冰

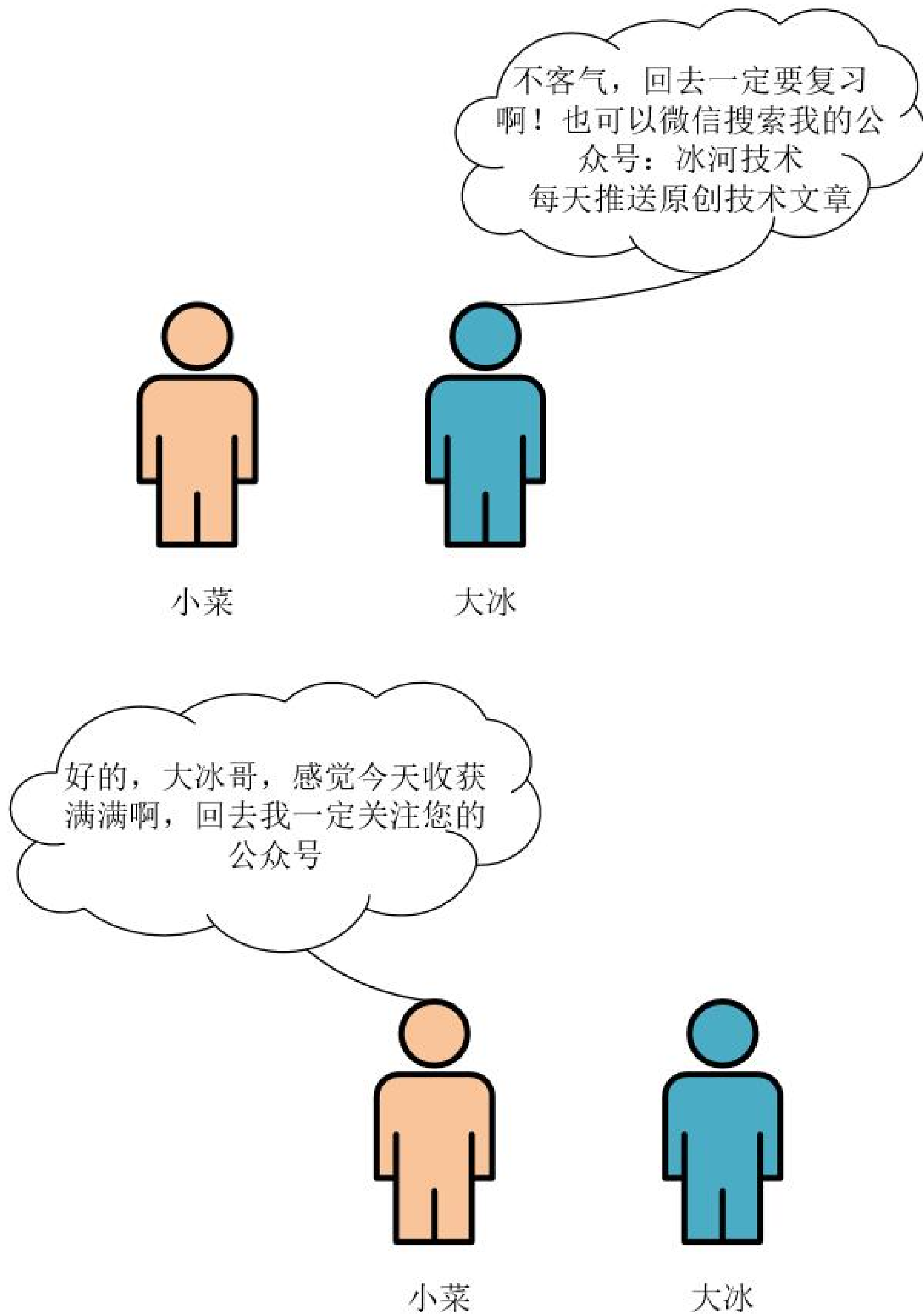
好的，大冰哥，谢谢您！



小菜



大冰



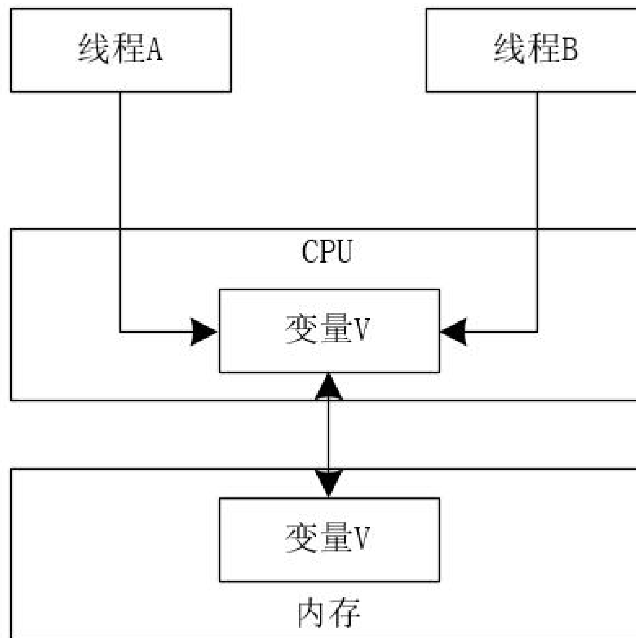
总结

了解并掌握并发编程问题的“幕后黑手”，有助于我们更好的学习并发知识和解决并发问题。

【源头一】缓存导致的可见性问题

可见性就是说一个线程对共享变量的修改，另一个线程能够立刻看到。

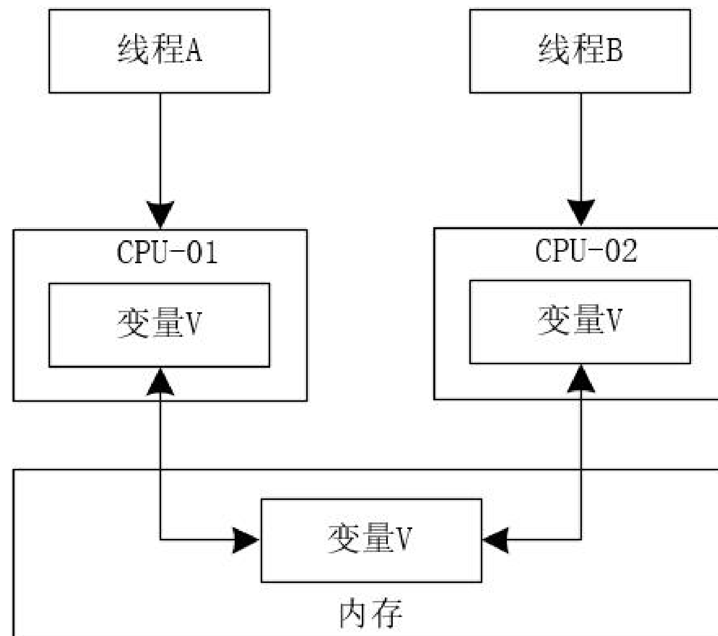
在CPU单核时代，操作系统上所有的线程都是运行在同一个CPU上，操作同一个CPU的缓存。一个线程对缓存的写，对另外一个线程一定可见。我们可以简单的用下图来表示。



由上图我们可以看出，由于只有一个CPU内核，线程A和线程B无论谁修改了CPU中的变量V，另一个线程读到的变量V一定是最新的值。

也就是说，在单核CPU中，不存在线程的可见性问题。大家可以记住这个结论。

在多核CPU中上述的结论就不成立了。因为在多核CPU中，每个CPU内核都有自己的缓存。当多个线程在不同的CPU核心上运行时，这些线程操作的是不同的CPU缓存。一个线程对缓存的写，对另外一个线程不一定可见。我们可以将多线程在多核CPU上修改变量的过程总结成如下图所示。



由上图我们可以看出，由于CPU是多核的，线程A操作的是CPU-01上的缓存，线程B操作的是CPU-02上的缓存，此时，线程A对变量V的修改对线程B是不可见的，反之亦然。

这里，我们可以使用一段Java代码来验证多线程的可见性问题。在下面的代码中，定义了一个long类型的成员变量count。有一个名称为addCount的方法，这个方法中对count的值累加1000次。同时，execute方法中，启动两个线程，分别调用addCount方法，等待两个线程执行后，返回count的值。代码如下所示。

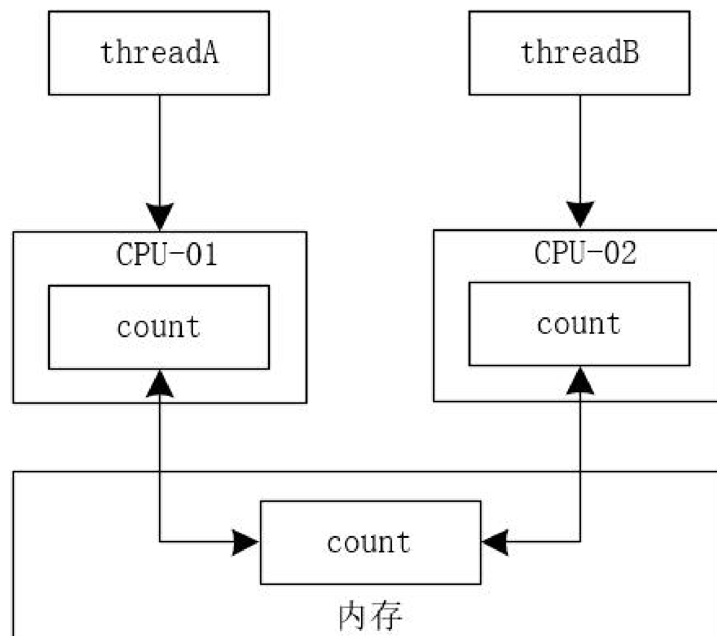
```
package io.binghe.concurrent.lab01;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试线程的可见性
 */
public class TestThread {
```

```java
    private long count = 0;

    //对count的值累加1000次
    private void addCount(){
        for(int i = 0; i < 1000; i++){
            count ++;
        }
    }

    public long execute() throws InterruptedException {
        //创建两个线程，执行count的累加操作
        Thread threadA = new Thread(() ->{
           addCount();
        });
        Thread threadB = new Thread(() ->{
           addCount();
        });
        //启动线程
        threadA.start();
        threadB.start();

        //等待线程执行结束
        threadA.join();
        threadB.join();

        //返回结果
        return count;
    }

    public static void main(String[] args) throws InterruptedException {
        TestThread testThread = new TestThread();
        long count = testThread.execute();
        System.out.println(count);
    }
}
```

在同一个线程中，连续调用两次addCount方法，count变量的值就是2000。但是实际上在两个线程中，调用addCount方法时，count的值最终是一个介于1000到2000之间的随机数。

我们一起来分析下这种情况：假设线程A和线程B同时执行，第一次都会将count = 0读取到各自的CPU缓存中，执行完count++之后，各自CPU缓存中的count的值为1。线程A和线程B同时将count写入内存后，内存中的count值为1，而不是2。在整个过程中，线程A和线程B都是基于各自CPU中缓存的count的值来进行计算的，最终会导致count的值小于或者等于2000。这就是缓存导致的可见性问题。

我们也可以使用如图来简单描述下线程A和线程B对于count变量的修改过程。



实际上，如果将上述的代码由循环1000次修改为循环1亿次，你会发现最终count的值会接近于1亿，而不是2亿。如果只是循环1000次，count的值就会接近于2000。不信，你自己可以运行尝试。造成这种结果的原因就是两个线程不是同时启动的，中间存在一个时间差。

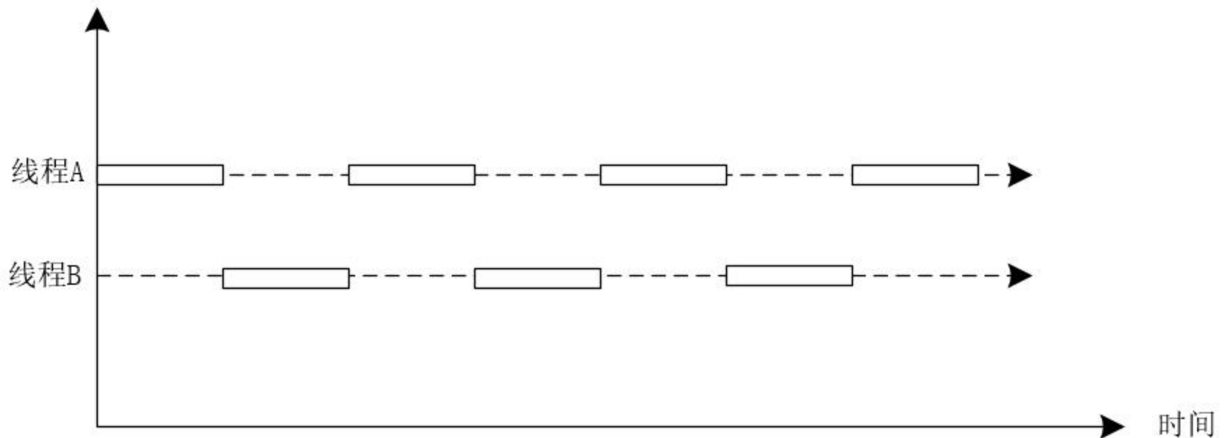
【源头二】线程切换带来的原子性问题

原子性是指一个或者多个操作在CPU中执行的过程不被中断的特性。

在现代操作系统中，一般都提供了多进程和多线程的功能。操作系统允许进程执行一小段时间，过了这段时间，操作系统就会重新选择一个进程来执行。我们称这种情况叫做任务切换，这一小段时间被称为时间片。

在如今的操作系统中，大部分的任务切换都是基于线程来执行的，我们也可以将任务切换叫作线程切换。

我们可以简单的使用下图来表示操作系统中的线程切换过程。



图中存在两个线程，分别为线程A和线程B，其中线程A和线程B中的每个小方块代表此时线程占有CPU并执行任务，每个虚线部分代表此时的线程不占用CPU资源。CPU会在线程A和线程B之间频繁切换。

Java并发程序是基于多线程来编写的，这也会涉及到CPU对任务的切换。正是CPU中对任务的切换机制，造成了并发编程中的第二个诡异的问题。

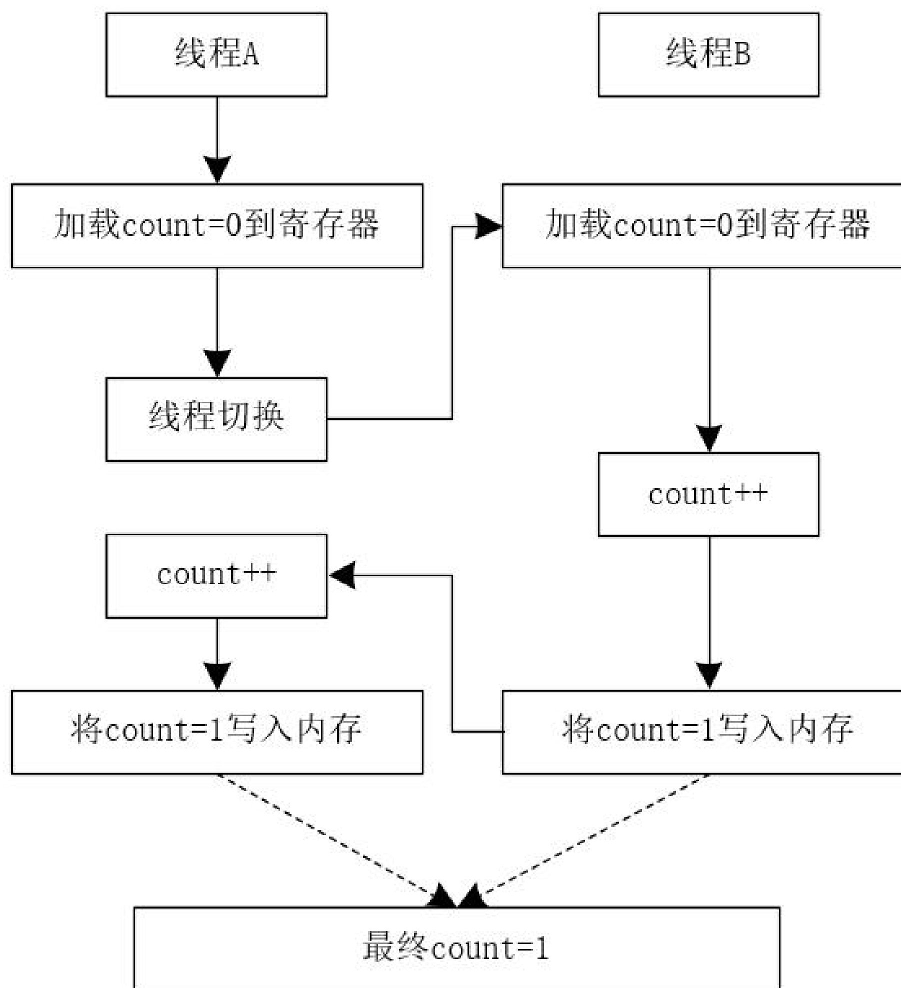
在并发编程中，往往Java中的一条简单的语句，对应着CPU中的多条指令。例如，我们编写的TestThread类中的如下代码。

```
count++
```

看似简单的一条count自增的代码，实际上对应着CPU中的多条指令。我们将CPU的指令简化成如下三步操作。

- 指令1：把变量count从内存加载的CPU寄存器。
- 指令2：在寄存器中执行count++操作。
- 指令3：将结果写入缓存（可能是CPU缓存，也可能是内存）。

在操作系统执行线程切换时，可能发生在任何一条CPU指令完成后，而不是程序中的某条语句完成后。如果线程A执行完指令1后，操作系统发生了线程切换，当两个线程都执行count++操作后，得到的结果是1而不是2。这里，我们可以使用下图来表示这个过程。



在上图中，线程A将count=0加载到CPU的寄存器后，发生了线程切换。此时内存中的count值仍然为0，线程B将count=0加载到寄存器，执行count++操作，并将count=1写到内存。此时，CPU切换到线程A，执行线程A中的count++操作后，线程A中的count值为1，线程A将count=1写入内存，此时内存中的count值最终为1。

综上：CPU能够保证的原子性是CPU指令级别的，而不是编程语言级别的。我们在编写高并发程序时，需要在编程语言级别保证程序的原子性。

【源头三】编译优化带来的有序性问题

有序性是指程序按照代码的既定顺序执行。

编译器或者解释器为了优化程序的执行性能，有时会改变程序的执行顺序。但是，编译器或者解释器对程序的执行顺序进行修改，可能会导致意想不到的问题！

在Java程序中，一个经典的案例就是使用双重检查机制来创建单例对象。例如，在下面的代码中，在getInstance()方法中获取对象实例时，首先判断instance对象是否为空，如果为空，则锁定当前类的class对象，并再次检查instance是否为空，如果instance对象仍然为空，则为instance对象创建一个实例。

```
package io.binghe.concurrent.lab01;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试单例
 */
public class SingleInstance {

    private static SingleInstance instance;

    public static SingleInstance getInstance(){
        if(instance == null){
            synchronized (SingleInstance.class){
                if(instance == null){
                    instance = new SingleInstance();
                }
            }
```

```
        }
        return instance;
    }
}
```

如果编译器或者解释器不会对上面的程序进行优化，整个代码的执行过程如下所示。

假设此时有线程A和线程B两个线程同时调用getInstance()方法来获取对象实例，两个线程会同时发现instance对象为空，此时会同时对SingleInstance.class加锁，而JVM会保证只有一个线程获取到锁，这里我们假设是线程A获取到锁。则线程B由于未获取到锁而进行等待。接下来，线程A再次判断instance对象为空，从而创建instance对象的实例，最后释放锁。此时，线程B被唤醒，线程B再次尝试获取锁，获取锁成功后，线程B检查此时的instance对象已经不再为空，线程B不再创建instance对象。

上面的一切看起来很完美，但是这一切的前提是编译器或者解释器没有对程序进行优化，也就是说CPU没有对程序进行重排序。而实际上，这一切都只是我们自己觉得是这样的。

在真正高并发环境下运行上面的代码获取instance对象时，创建对象的new操作会因为编译器或者解释器对程序的优化而出现问题。也就是说，问题的根源在于如下一行代码。

```
instance = new SingleInstance();
```

对于上面的一行代码来说，会有3个CPU指令与其对应。

1.分配内存空间。

2.初始化对象。

3.将instance引用指向内存空间。

正常执行的CPU指令顺序为1—>2—>3，CPU对程序进行重排序后的执行顺序可能为1—>3—>2。此时，就会出现问题。

当CPU对程序进行重排序后的执行顺序为1—>3—>2时，我们将线程A和线程B调用getInstance()方法获取对象实例的两种步骤总结如下所示。

【第一种步骤】

（1）假设线程A和线程B同时进入第一个if条件判断。

（2）假设线程A首先获取到synchronized锁，进入synchronized代码块，此时因为instance对象为null，所以，此时执行instance = new SingleInstance()语句。

（3）在执行instance = new SingleInstance()语句时，线程A会在JVM中开辟一块空白的内存空间。

（4）线程A将instance引用指向空白的内存空间，在没有进行对象初始化的时候，发生了线程切换，线程A释放synchronized锁，CPU切换到线程B上。

（5）线程B进入synchronized代码块，读取到线程A返回的instance对象，此时这个instance不为null，但是并未进行对象的初始化操作，是一个空对象。此时，线程B如果使用instance，就可能出现问题！！！

【第二种步骤】

（1）线程A先进入if条件判断，

（2）线程A获取synchronized锁，并进行第二次if条件判断，此时的instance为null，执行instance = new SingleInstance()语句。

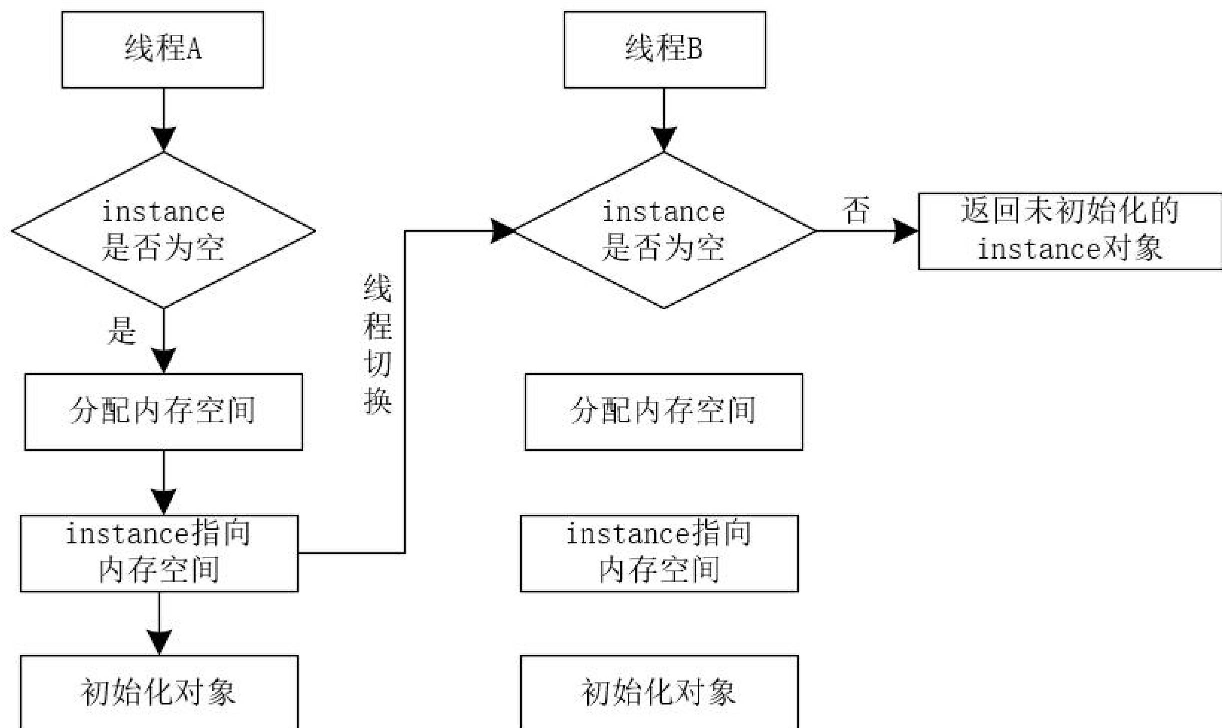
（3）线程A在JVM中开辟一块空白的内存空间。

（4）线程A将instance引用指向空白的内存空间，在没有进行对象初始化的时候，发生了线程切换，CPU切换到线程B上。

（5）线程B进行第一次if判断，发现instance对象不为null，但是此时的instance对象并未进行初始化操作，是一个空对象。如果线程B直接使用这个instance对象，就可能出现问题！！！

在第二种步骤中，即使发生线程切换时，线程A没有释放锁，则线程B进行第一次if判断时，发现instance已经不为null，直接返回instance，而无需尝试获取synchronized锁。

我们可以将上述过程简化成下图所示。



总结

我们在介绍多线程编程时，往往会介绍并发编程的三大特性：可见性、原子性和有序性。在并发编程领域中，对于各种问题的追本溯源我们可以总结出如下问题的根源。

- **缓存带来了可见性问题。**
- **线程切换带来了原子性问题。**
- **编译优化带来了有序性问题。**

我们只有深刻的理解了并发编程的问题源头，才能编写出更加健壮的高并发程序！！

解密诡异并发问题的第一个幕后黑手——可见性问题

写在前面

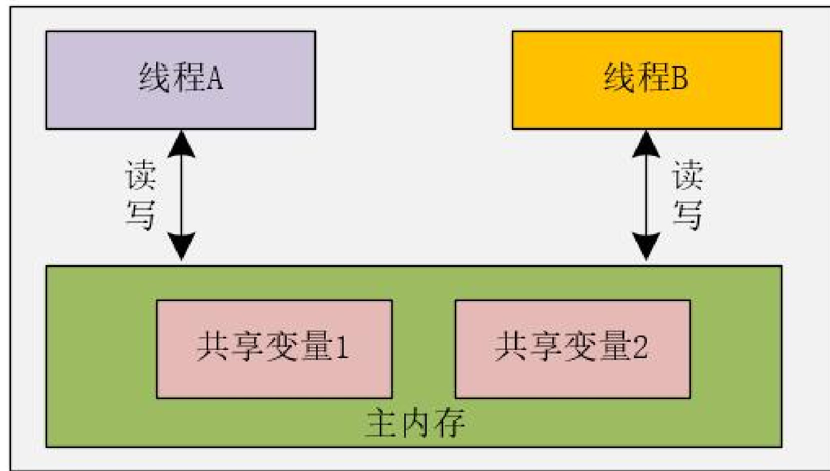
> 大冰：小菜童鞋，昨天讲解的内容复习了吗？
>
> 小菜：复习了，大冰哥。
>
> 大冰：那你说说我们昨天都讲了哪些内容呢？
>
> 小菜：昨天讲了并发编程的难点，由这些难点引出我们需要了解导致这些问题的"幕后黑手"。对于并发编程来说，计算机和操作系统的制作商为了提升计算机和系统的性能，为CPU增加了缓存，为操作系统增加了进程和线程，优化了CPU指令的执行顺序。而这些优化措施恰恰是导致并发编程频繁出现诡异问题的根源。
>
> 大冰：很好，小菜童鞋，掌握的不错，今天，我们就深入讲讲由缓存导致的可见性问题，这就是并发问题的三大"幕后黑手"之一，这个知识点非常重要，好好听。

可见性

> 对于什么是可见性，比较官方的解释就是：一个线程对共享变量的修改，另一个线程能够立刻看到。

说的直白些，就是两个线程共享一个变量，无论哪一个线程修改了这个变量，则另外的一个线程都能够看到上一个线程对这个变量的修改。这里的共享变量，指的是多个线程都能够访问和修改这个变量的值，那么，这个变量就是共享变量。

例如，线程A和线程B，它们都是直接修改主内存中的共享变量，无论是线程A修改了共享变量，还是线程B修改了共享变量，则另一个线程从主内存中读取出来的变量值，一定是修改过的值，这就是线程的可见性。



可见性问题

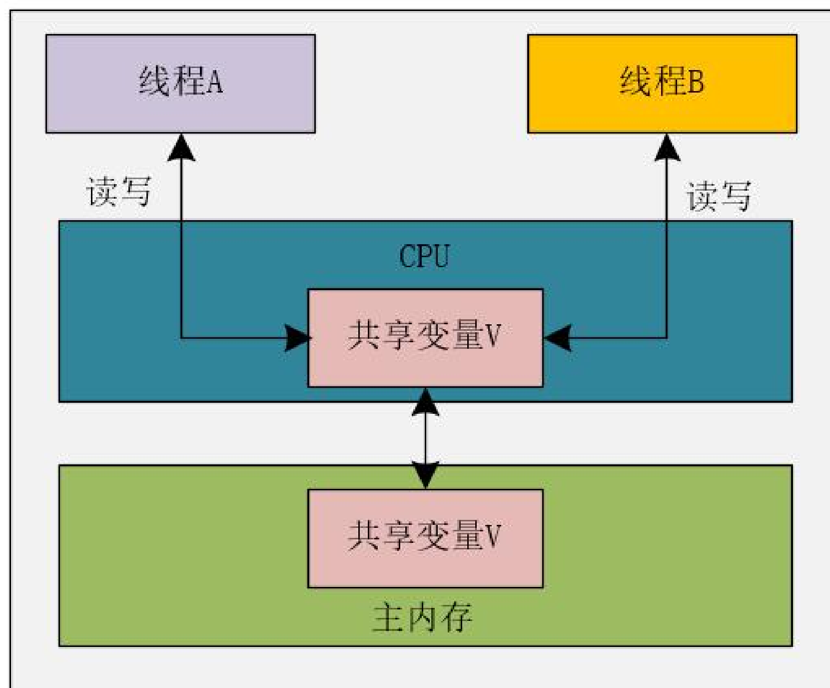
> 可见性问题，可以这样理解：一个线程修改了共享变量，另一个线程不能立刻看到，这是由CPU添加了缓存导致的问题。

理解了什么是可见性，再来看可见性问题就比较好理解了。既然可见性是一个线程修改了共享变量后，另一个线程能够立刻看到对共享变量的修改，如果不能立刻看到，这就会产生可见性的问题。

单核CPU不存在可见性问题

理解可见性问题我们还需要注意一点，那就是**在单核CPU上不存在可见性问题。**这是为什么呢?

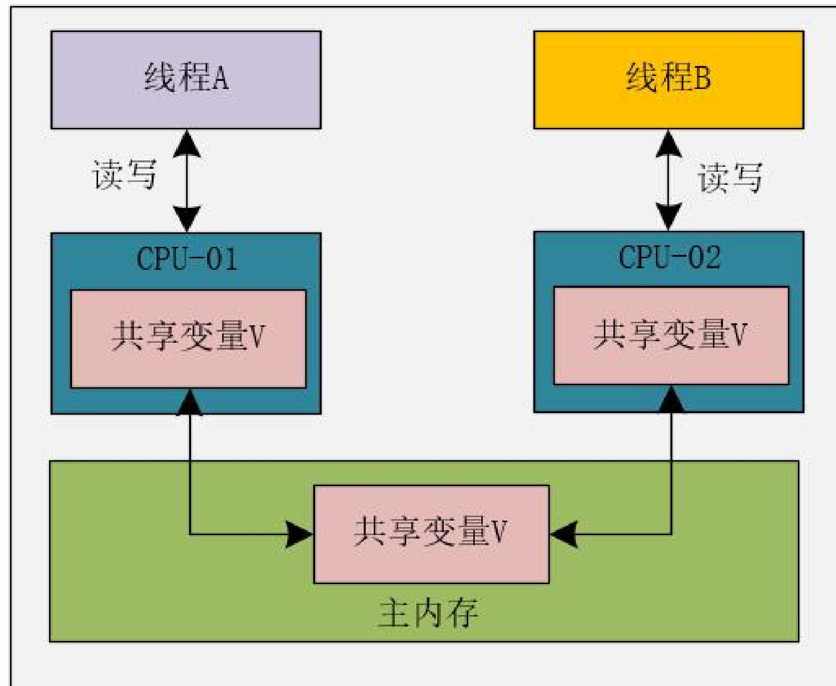
因为**在单核CPU上，无论创建了多少个线程，同一时刻只会有一个线程能够获取到CPU的资源来执行任务**，即使这个单核的CPU已经添加了缓存。这些线程都是运行在同一个CPU上，操作的是同一个CPU的缓存，只要其中一个线程修改了共享变量的值，那另外的线程就一定能够访问到修改后的变量值。



多核CPU存在可见性问题

单核CPU由于同一时刻只会有一个线程执行，而每个线程执行的时候操作的都是同一个CPU的缓存，所以，单核CPU不存在可见性问题。但是到了多核CPU上，就会出现可见性问题了。

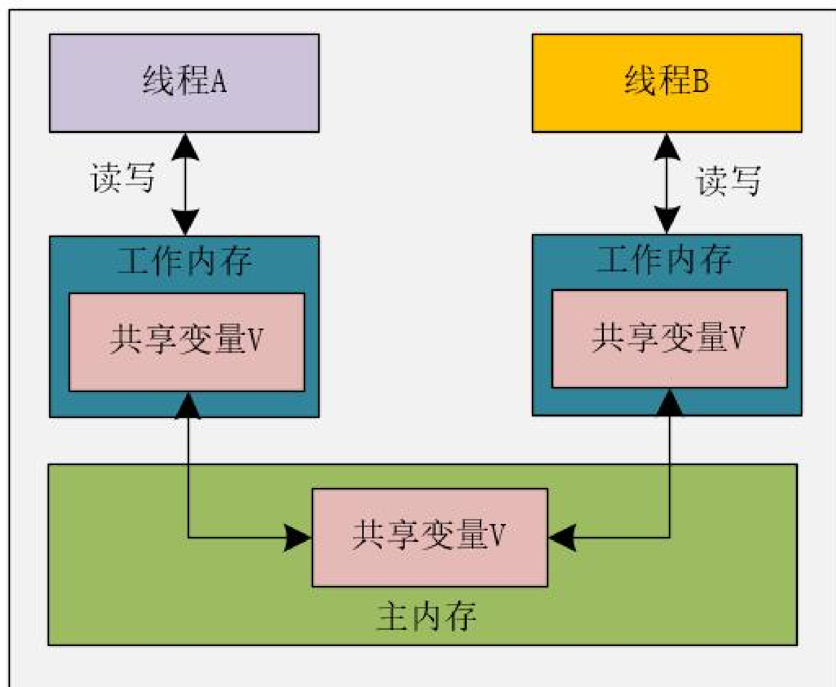
这是因为在多核CPU上，每个CPU的内核都有自己的缓存。当多个不同的线程运行在不同的CPU内核上时，这些线程操作的是不同的CPU缓存。一个线程对其绑定的CPU的缓存的写操作，对于另外一个线程来说，不一定是可见的，这就造成了线程的可见性问题。



例如，上面的图中，由于CPU是多核的，线程A操作的是CPU-01上的缓存，线程B操作的是CPU-02上的缓存，此时，线程A对变量V的修改对线程B是不可见的，反之亦然。

Java中的可见性问题

使用Java语言编写并发程序时，如果线程使用变量时，会把主内存中的数据复制到线程的私有内存，也就是工作内存中，每个线程读写数据时，都是操作自己的工作内存中的数据。



此时，Java中线程读写共享变量的模型与多核CPU类似，原因是Java并发程序运行在多核CPU上时，线程的私有内存，也就是工作内存就相当于多核CPU中每个CPU内核的缓存了。

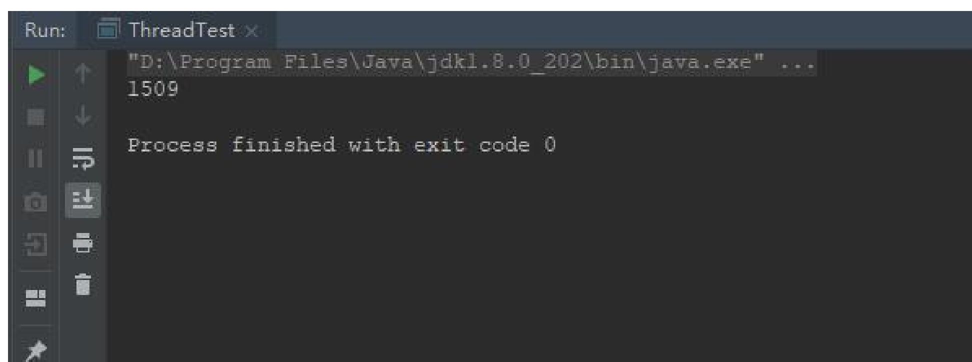
由上图，同样可以看出，线程A对共享变量的修改，线程B不一定能够立刻看到，这也就会造成可见性的问题。

代码示例

我们使用一个Java程序来验证多线程的可见性问题，在这个程序中，定义了一个long类型的成员变量count，有一个名称为addCount的方法，这个方法中对count的值进行加1操作。同时，在execute方法中，分别启动两个线程，每个线程调用addCount方法1000次，等待两个线程执行完毕后，返回count的值，代码如下所示。

```
package io.mykit.concurrent.lab01;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试可见性
 */
public class ThreadTest {

    private long count = 0;

    private void addCount(){
        count ++;
    }

    public long execute() throws InterruptedException {
        Thread threadA = new Thread(() -> {
            for(int i = 0; i < 1000; i++){
                addCount();
            }
        });

        Thread threadB = new Thread(() -> {
            for(int i = 0; i < 1000; i++){
                addCount();
            }
        });

        //启动线程
        threadA.start();
        threadB.start();

        //等待线程执行完成
        threadA.join();
        threadB.join();
        return count;
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadTest threadTest = new ThreadTest();
        long count = threadTest.execute();
        System.out.println(count);
    }
}
```

我们运行下这个程序，结果如下图所示。



可以看到这个程序的结果是1509，而不是我们期望的2000。这是为什么呢？让我们一起来分析下这个程序。

首先，变量count属于ThreadTest类的成员变量，这个成员变量对于线程A和线程B来说，是一个共享变量。假设线程A和线程B同时执行，它们同时将count=0读取到各自的工作内存中，每个线程第一次执行完count++操作后，同时将count的值写入内存，此时，内存中count的值为1，而不是我们想象的2。而在整个计算的过程中，线程A和线程B都是基于各自工作内存中的count值进行计算。这就导致了最终的count值小于2000。

归根结底：可见性的问题是CPU的缓存导致的。

总结

可见性是一个线程对共享变量的修改，另一个线程能够立刻看到，如果不能立刻看到，就可能会产生可见性问题。在单核CPU上是不存在可见性问题的，可见性问题主要存在于运行在多核CPU上的并发程序。归根结底，可见性问题还是由CPU的缓存导致的，而缓存导致的可见性问题是导致诸多诡异的并发编程问题的"幕后黑手"之一。我们只有深入理解了缓存导致的可见性问题，并在实际工作中时刻注意避免可见性问题，才能更好的编写出高并发程序。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

结尾

> 大冰：这就是今天我们讲的第一个"幕后黑手"——缓存导致的可见性问题，小菜童鞋，今天讲的知识干货比较多，你可能听一遍不是很懂，回去后一定要认真复习啊！
>
> 小菜：好的，大冰哥。今天讲的内容确实都是干货啊，我回去要多看几遍才行啊！

解密导致并发问题的第二个幕后黑手——原子性问题

写在前面

> 大冰：小菜童鞋，昨天讲解的内容复习了吗？
>
> 小菜：复习了大冰哥，昨天的内容干货满满啊，感觉自己收获很大。
>
> 大冰：那你说说昨天都讲了哪些内容呢？
>
> 小菜：昨天主要讲了线程的可见性和可见性问题。可见性是指一个线程对共享变量的修改，另一个线程能够立刻看到，如果不能立刻看到，就可能会产生可见性问题。在单核CPU上是不存在可见性问题的，可见性问题主要存在于运行在多核CPU上的并发程序。归根结底，可见性问题还是由CPU的缓存导致的，而缓存导致的可见性问题是导致诸多诡异的并发编程问题的"幕后黑手"之一。
>
> 大冰：很好，小菜童鞋，复习的不错，今天，我们继续讲并发问题的第二个"幕后黑手"——线程切换带来的原子性问题，这个知识点也是非常重要的，一定要好好听。

原子性

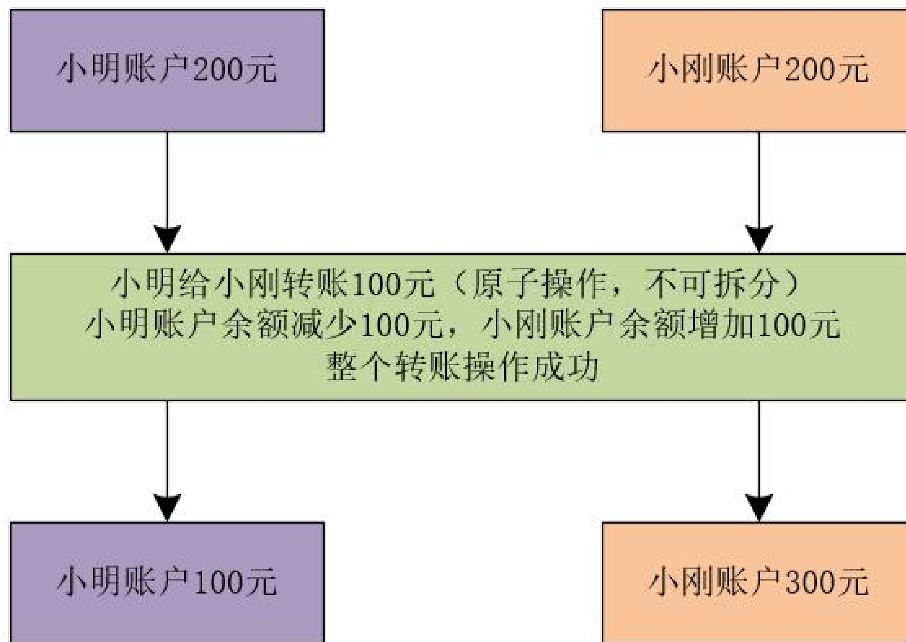
> 原子性是指一个或者多个操作在CPU中执行的过程不被中断的特性。原子性操作一旦开始运行，就会一直到运行结束为止，中间不会有中断的情况发生。

我们也可以这样理解原子性，就是线程在执行一系列操作时，这些操作会被当做一个不可拆分的整体执行，这些操作要么全部执行，要么全部不执行，不会存在只执行一部分的情况，这就是原子性操作。

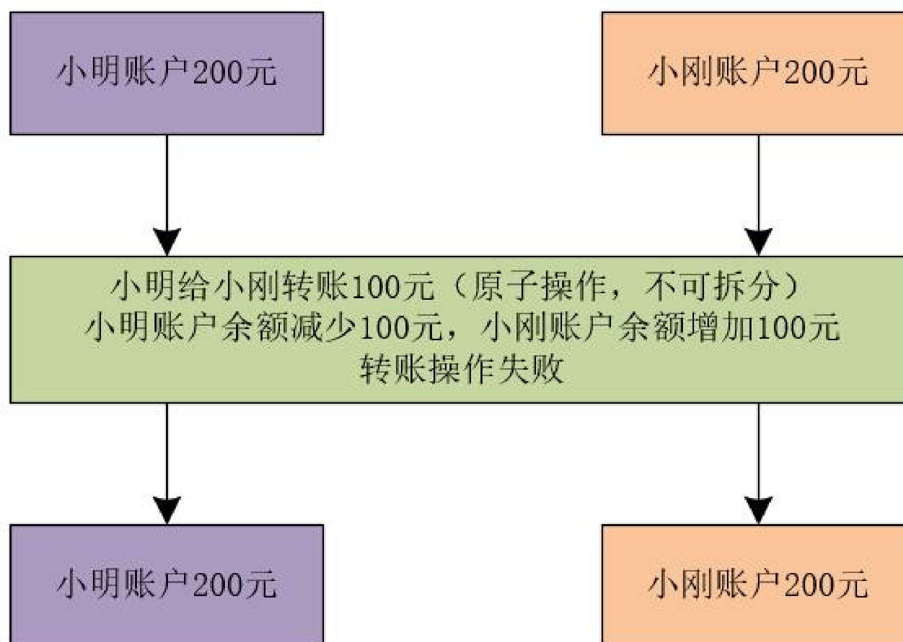
关于原子性操作一个**典型的场景就是转账**，例如，小明和小刚的账户余额都是200元，此时小明给小刚转账100元，如果转账成功，则小明的账户余额为100元，小刚的账户余额为300元；如果转账失败，则小明和小刚的账户余额仍然为200元。不会存在小明账户为100元，小刚账户为200元，或者小明账户为200元，小刚账户为300元的情况。

这里，小明给小刚转账100元的操作，就是一个原子性操作，它涉及小明账户余额减少100元，小刚账户余额增加100元的操作，这两个操作是一个不可分割的整体，要么全部执行，要么全部不执行。

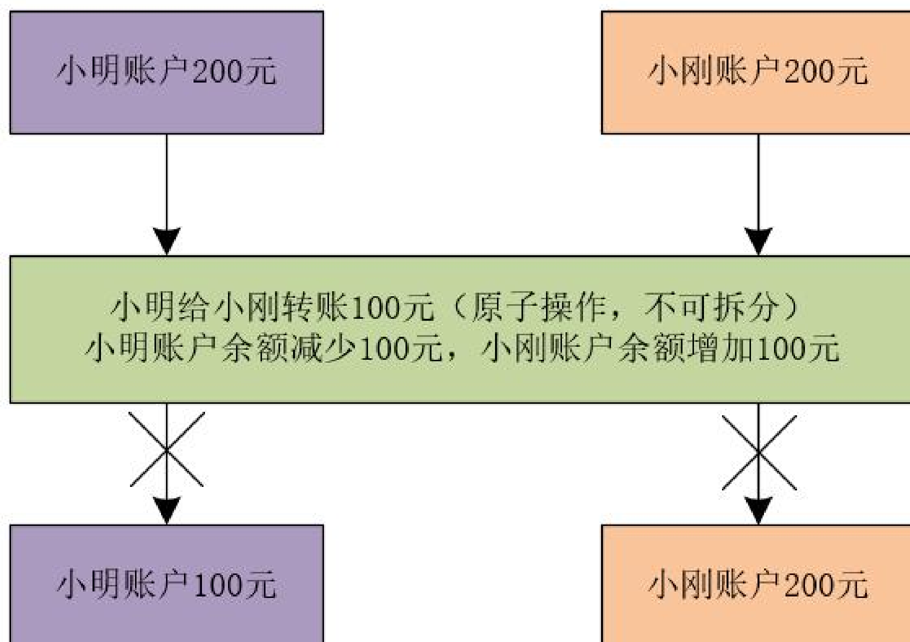
小明给小刚转账成功，则如下所示。



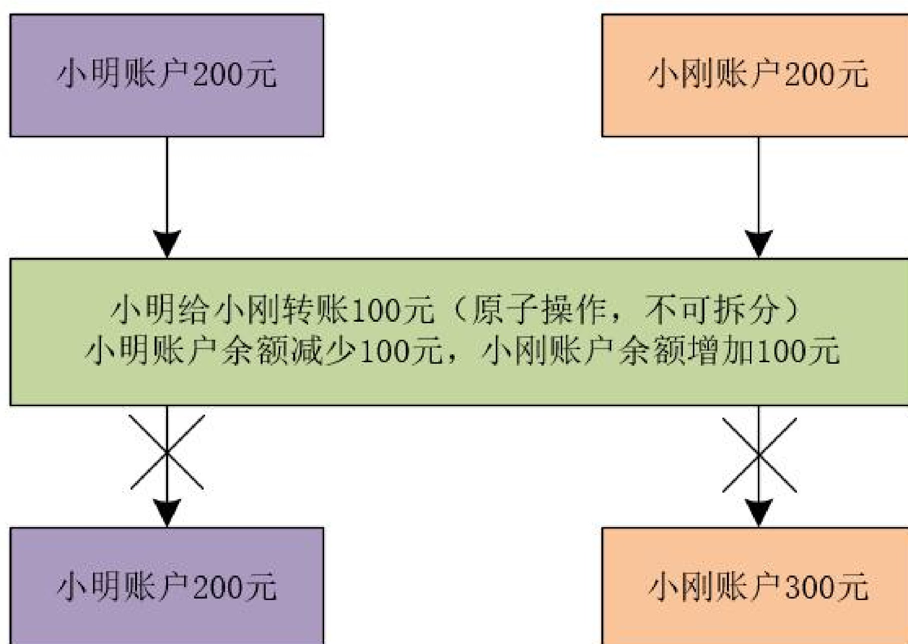
小明给小刚转账失败，则如下所示。



不会出现小明账户为100元，小刚账户为200元的情况。



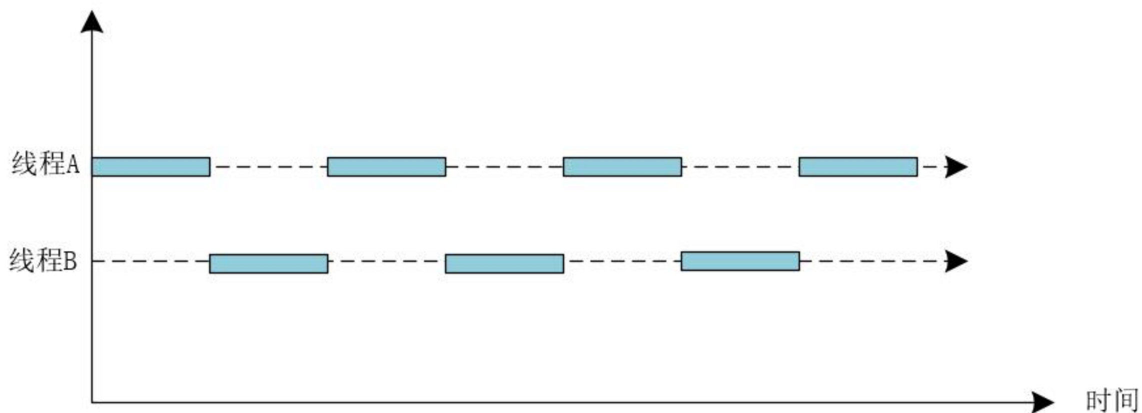
也不会出现小明账户为200元，小刚账户为300元的情况。



线程切换

> 在并发编程中，往往设置的线程数目会大于CPU数目，而每个CPU在同一时刻只能被一个线程使用。而CPU资源的分配采用了时间片轮转策略，也就是给每个线程分配一个时间片，线程在这个时间片内占用CPU的资源来执行任务。当占用CPU资源的线程执行完任务后，会让出CPU的资源供其他线程运行，这就是任务切换，也叫做线程切换或者线程的上下文切换。

如果大家还是不太理解的话，我们可以用下面的图来模拟线程在CPU中的切换过程。



在图中存在线程A和线程B两个线程，其中线程A和线程B中的每个小方块代表此时线程占有CPU资源并执行任务，这个小方块占有的时间，被称为时间片，在这个时间片中，占有CPU资源的线程会在CPU上执行，未占有CPU资源的线程则不会在CPU上执行。而每个虚线部分就代表了此时的线程不占用CPU资源。CPU会在线程A和线程B之间频繁切换。

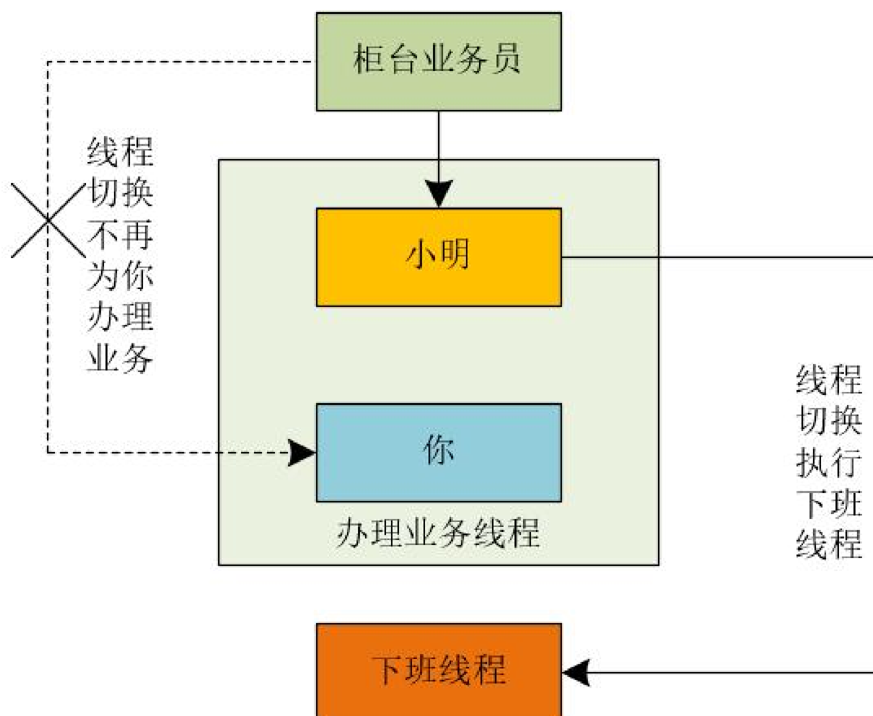
原子性问题

理解了什么是原子性，再看什么是原子性问题就比较简单了。

> 原子性问题是指一个或者多个操作在CPU中执行的过程中出现了被中断的情况。

线程在执行某项操作时，此时由于CPU发生了线程切换，CPU转而去执行其他的任务，中断了当前线程执行的操作，这就会造成原子性问题。

如果你还不能理解的话，我们来举一个例子：假设你在银行排队办理业务，小明在你前面，柜台的业务员为小明办理完业务，正好排到你时，此时银行下班了，柜台的业务员微笑着告诉你：实在不好意思，先生（女士），我们下班了，您明天再来吧！此时的你就好比是正好占有了CPU资源的线程，而柜台的业务员就是那颗发生了线程切换的CPU，她将线程切换到了下班这个线程，执行下班的操作去了。



Java中的原子性问题

在Java中，并发程序是基于多线程技术来编写的，这也会涉及到CPU的对于线程的切换问题，**正是CPU中对任务的切换机制，导致了并发编程会出现原子性的诡异问题，而原子性问题，也成为了导致并发问题的第二个“幕后黑手”。**

在并发编程中，往往Java语言中一条简单的语句，会对应着CPU中的多条指令，假设我们编写的ThreadTest类的代码如下所示。

```java
package io.mykit.concurrent.lab01;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试原子性
 */
public class ThreadTest {

    private Long count;

    public Long getCount(){
        return count;
    }

    public void incrementCount(){
        count++;
    }
}
```

接下来，我们打开ThreadTest类的class文件所在的目录，在cmd命令行输入如下命令。

```
javap -c ThreadTest
```

得出如下的结果信息，如下所示。

```
d:>javap -c ThreadTest
Compiled from "ThreadTest.java"
public class io.mykit.concurrent.lab01.ThreadTest {
  public io.mykit.concurrent.lab01.ThreadTest();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public java.lang.Long getCount();
    Code:
       0: aload_0
       1: getfield      #2                  // Field count:Ljava/lang/Long;
       4: areturn

  public void incrementCount();
    Code:
       0: aload_0
       1: getfield      #2                  // Field count:Ljava/lang/Long;
       4: astore_1
       5: aload_0
       6: aload_0
       7: getfield      #2                  // Field count:Ljava/lang/Long;
      10: invokevirtual #3                  // Method java/lang/Long.longValue:()J
      13: lconst_1
      14: ladd
      15: invokestatic  #4                  // Method java/lang/Long.valueOf:(J)Ljava/lang/Long;
      18: dup_x1
      19: putfield      #2                  // Field count:Ljava/lang/Long;
      22: astore_2
      23: aload_1
      24: pop
      25: return
}
```

这里，我们主要关注下incrementCount()方法对应的CPU指令，如下所示。

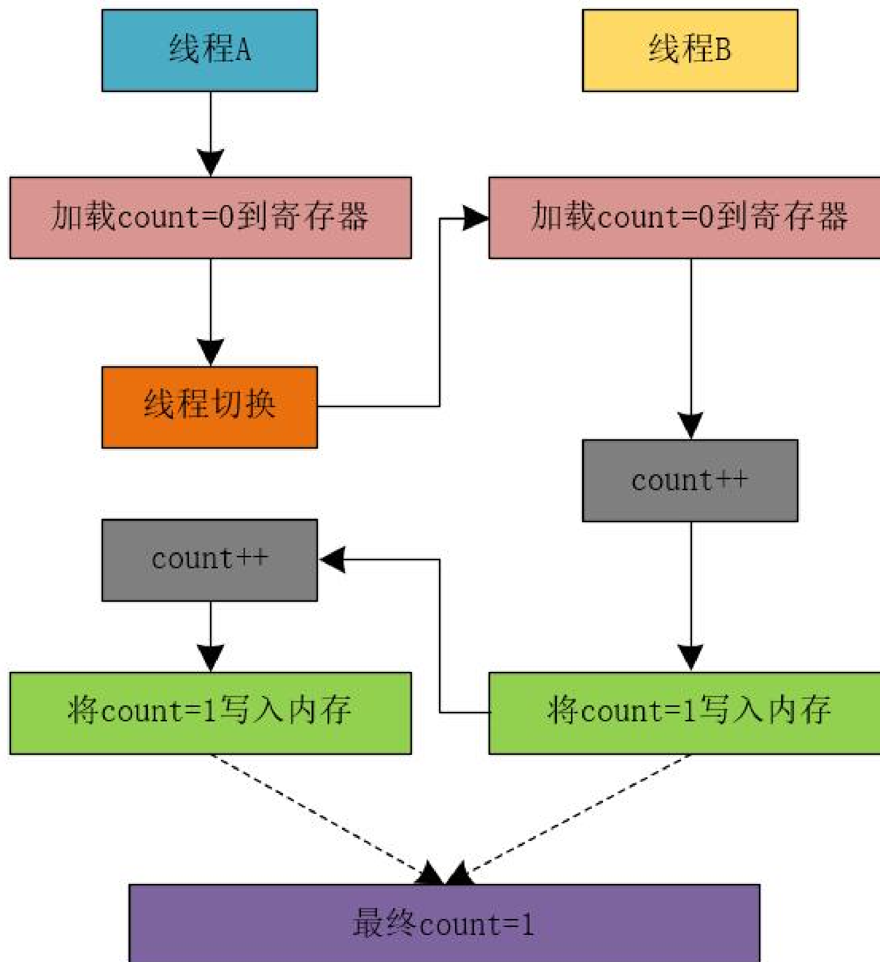
```
public void incrementCount();
    Code:
       0: aload_0
       1: getfield      #2                  // Field count:Ljava/lang/Long;
       4: astore_1
       5: aload_0
```

```
         6: aload_0
         7: getfield      #2                  // Field count:Ljava/lang/Long;
        10: invokevirtual #3                  // Method java/lang/Long.longValue:()J
        13: lconst_1
        14: ladd
        15: invokestatic  #4                  // Method java/lang/Long.valueOf:(J)Ljava/lang/Long;
        18: dup_x1
        19: putfield      #2                  // Field count:Ljava/lang/Long;
        22: astore_2
        23: aload_1
        24: pop
        25: return
```

我们看到，Java语言中短短的几行incrementCount()方法竟然对应着那么多的CPU指令。这些CPU指令我们大致可以分成三步。

- 指令1：把变量count从内存加载的CPU寄存器。
- 指令2：在寄存器中执行count++操作。
- 指令3：将结果写入缓存（可能是CPU缓存，也可能是内存）。

在操作系统执行线程切换时，可能发生在任何一条CPU指令完成后，而不是程序中的某条语句完成后。如果线程A执行完指令1后，操作系统发生了线程切换，当两个线程都执行count++操作后，得到的结果是1而不是2。这里，我们可以使用下图来表示这个过程。



由上图，我们可以看出：线程A将count=0加载到CPU的寄存器后，发生了线程切换。此时内存中的count值仍然为0，线程B将count=0加载到寄存器，执行count++操作，并将count=1写到内存。此时，CPU切换到线程A，执行线程A中的count++操作后，线程A中的count值为1，线程A将count=1写入内存，此时内存中的count值最终为1。

所以，如果在CPU中存在正在执行的线程，恰好此时CPU发生了线程切换，则可能会导致原子性问题，这也是导致并发编程频繁出问题的根源之一。我们只有充分理解并掌握线程的原子性以及引起原子性问题的根源，并在日常工作中时刻注意编写的并发程序是否存在原子性问题，才能更好的编写出并发程序。

总结

缓存带来的可见性问题、线程切换带来的原子性问题和编译优化带来的有序性问题，是导致并发编程频繁出现诡异问题的三个源头，我们已经介绍了缓存带来的可见性问题和线程切换带来的原子性问题。下一篇中，我们继续深耕高并发中的有序性问题。

写在最后

> 大冰：好了，今天就是我们讲的主要内容了，今天的内容同样最重要，回去后要好好复习。
>
> 小菜：好的，大冰哥，一定好好复习。

文末福利

> 微信搜索并关注「 **冰河技术** 」微信公众号，发送「 **jvm** 」领取全套《JVM指令手册》。

解密导致并发问题的第三个幕后黑手——有序性问题

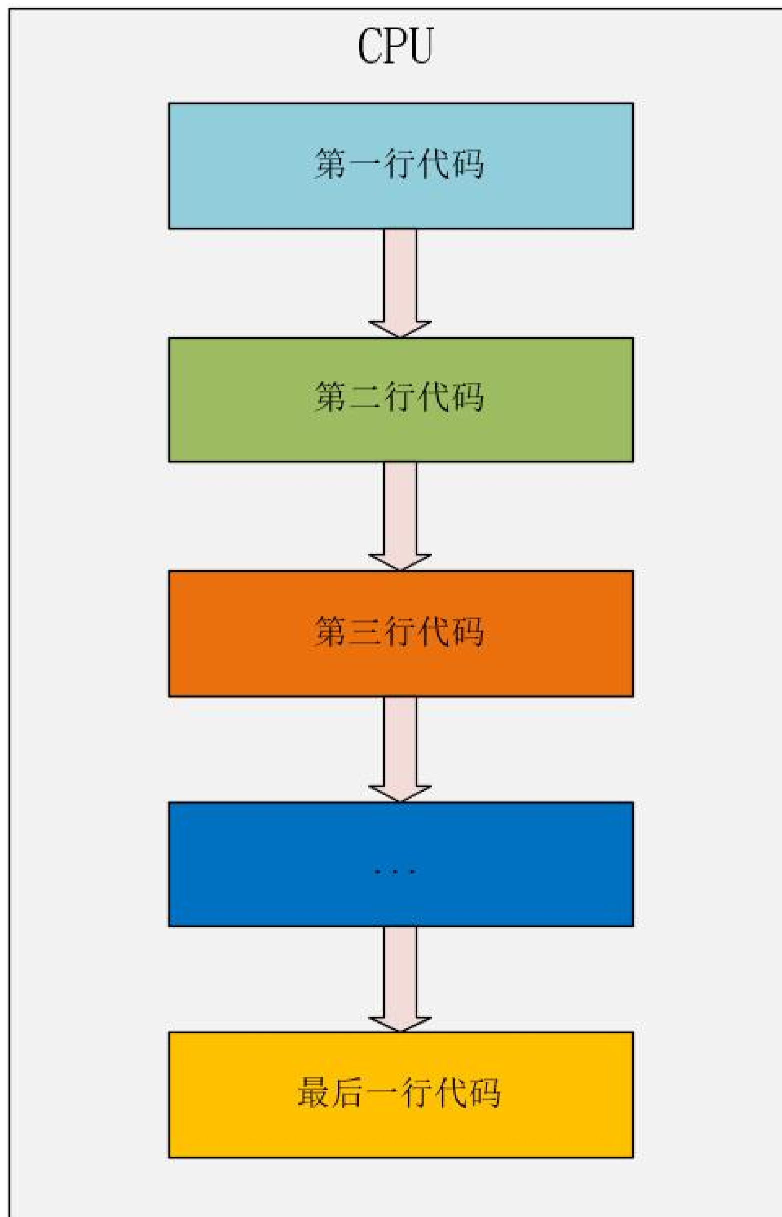
写在前面

> 大冰：小菜童鞋，昨天的内容复习了吗？
>
> 小菜：复习了大冰哥，昨天的内容干货满满啊，感觉自己收获很大。
>
> 大冰：那你说说昨天都讲了哪些内容呢？
>
> 小菜：昨天主要讲了原子性、线程切换和原子性问题，在编程语言中的一条语句可能会对应CPU中的多条指令，而CPU只能保证指令级别的原子性，不能保证编程语言级别的原子性，我们在编写并发程序时，需要自行确保编程语言级别语句的原子性。
>
> 大冰：很好，小菜童鞋，理解的不错，今天我们就来学习下引起并发编程各种诡异Bug的最后一个"幕后黑手"，也是最后一个引起并发编程Bug的源头。

有序性

> 有序性是指：按照代码的既定顺序执行。

说的通俗一点，就是代码会按照指定的顺序执行，例如，按照程序编写的顺序执行，先执行第一行代码，再执行第二行代码，然后是第三行代码，以此类推。如下图所示。

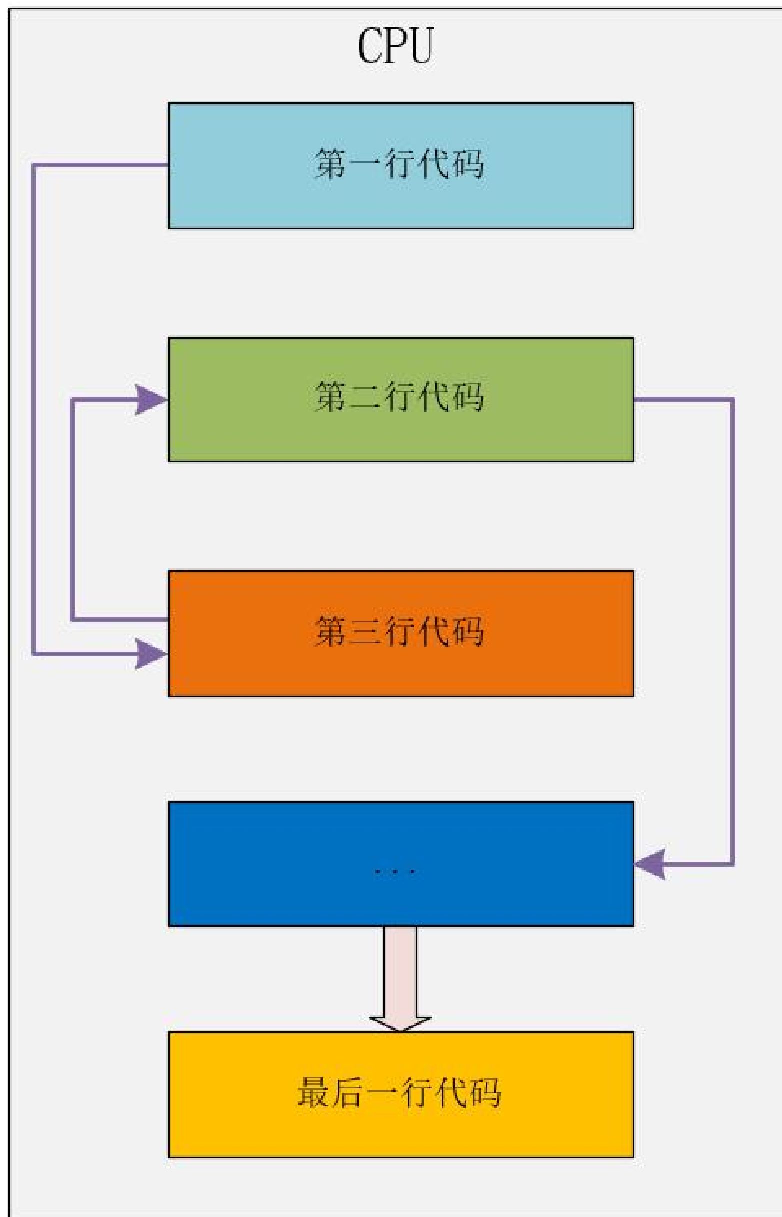


指令重排序

> 编译器或者解释器为了优化程序的执行性能，有时会改变程序的执行顺序。但是，编译器或者解释器对程序的执行顺序进行修改，可能会导致意想不到的问题！

在单线程下，指令重排序可以保证最终执行的结果与程序顺序执行的结果一致，但是在多线程下就会存在问题。

如果发生了指令重排序，则程序可能先执行第一行代码，再执行第三行代码，然后执行第二行代码，如下所示。



例如下面的三行代码。

```
int x = 1;
int y = 2;
int z = x + y;
```

CPU发生指令重排序时，能够保证x=1和y = 2这两行代码在z = x + y这行代码的上面，而x = 1和 y = 2的顺序就不一定了。在单线程下不会出现问题，但是在多线程下就不一定了。

有序性问题

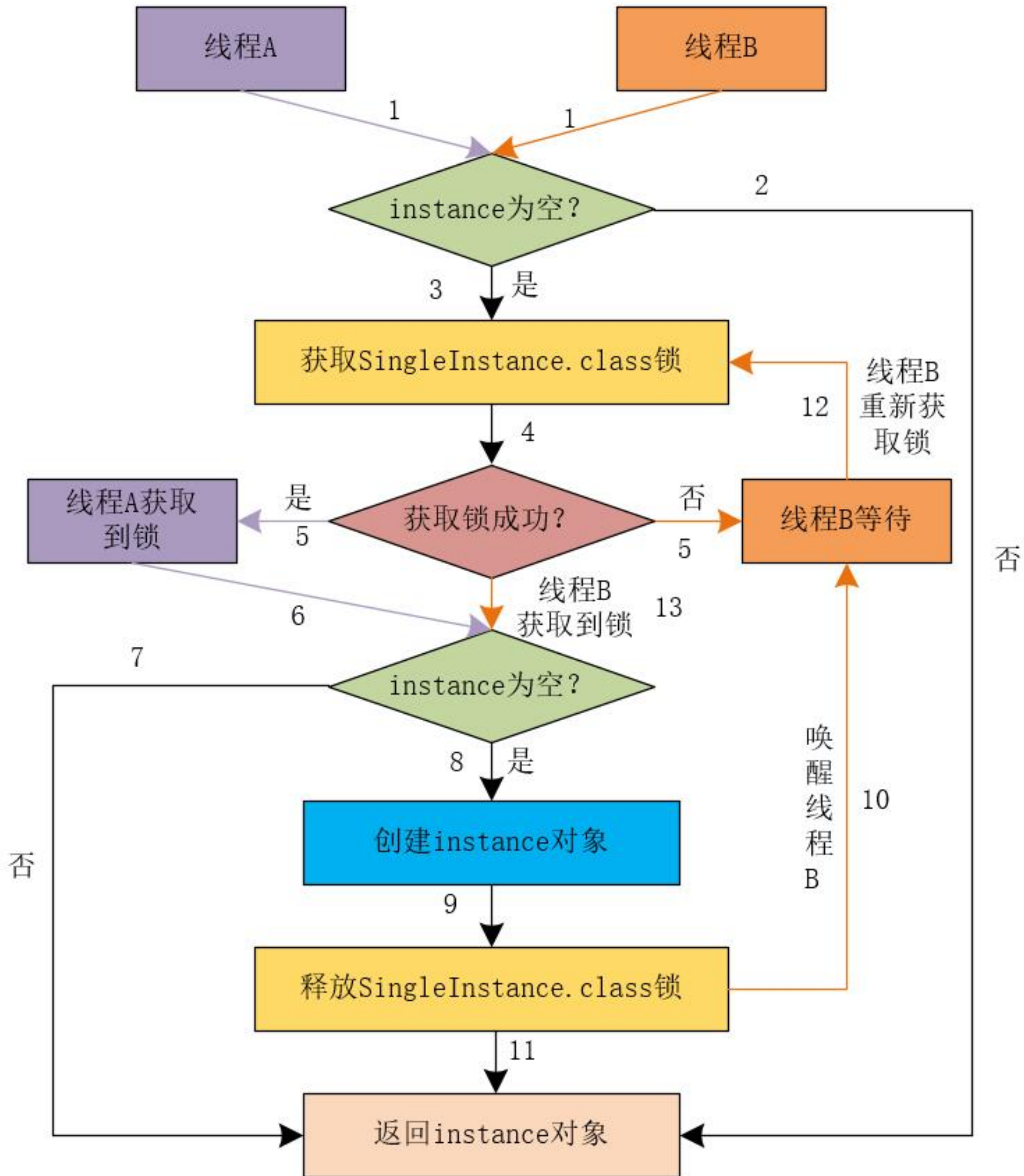
> CPU为了对程序进行优化，会对程序的指令进行重排序，此时程序的执行顺序和代码的编写顺序不一定一致，这就可能会引起有序性问题。

在Java程序中，一个经典的案例就是使用双重检查机制来创建单例对象。例如，在下面的代码中，在getInstance()方法中获取对象实例时，首先判断instance对象是否为空，如果为空，则锁定当前类的class对象，并再次检查instance是否为空，如果instance对象仍然为空，则为instance对象创建一个实例。

```
package io.binghe.concurrent.lab01;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试单例
 */
public class SingleInstance {
```

```java
    private static SingleInstance instance;

    public static SingleInstance getInstance(){
        if(instance == null){
            synchronized (SingleInstance.class){
                if(instance == null){
                    instance = new SingleInstance();
                }
            }
        }
        return instance;
    }
}
```

如果编译器或者解释器不会对上面的程序进行优化，整个代码的执行过程如下所示。



注意：为了让大家更加明确流程图的执行顺序，我在上图中标注了数字，以明确线程A和线程B执行的顺序。

假设此时有线程A和线程B两个线程同时调用getInstance()方法来获取对象实例，两个线程会同时发现instance对象为空，此时会同时对SingleInstance.class加锁，而JVM会保证只有一个线程获取到锁，这里我们假设是线程A获取到锁。则线程B由于未获取到锁而进行等待。接下来，线程A再次判断instance对象为空，从而创建instance对象的实例，最后释放锁。此时，线程B被唤醒，线程B再次尝试获取锁，获取锁成功后，线程B检查此时的instance对象已经不再为空，线程B不再创建instance对象。

上面的一切看起来很完美，但是这一切的前提是编译器或者解释器没有对程序进行优化，也就是说CPU没有对程序进行重排序。而实际上，这一切都只是我们自己觉得是这样的。

在真正高并发环境下运行上面的代码获取instance对象时，创建对象的new操作会因为编译器或者解释器对程序的优化而出现问题。也就是说，问题的根源在于如下一行代码。

```
instance = new SingleInstance();
```

对于上面的一行代码来说，会有3个CPU指令与其对应。

1.分配内存空间。

2.初始化对象。

3.将instance引用指向内存空间。

正常执行的CPU指令顺序为1—>2—>3，CPU对程序进行重排序后的执行顺序可能为1—>3—>2。此时，就会出现问题。

当CPU对程序进行重排序后的执行顺序为1—>3—>2时，我们将线程A和线程B调用getInstance()方法获取对象实例的两种步骤总结如下所示。

【第一种步骤】

（1）假设线程A和线程B同时进入第一个if条件判断。

（2）假设线程A首先获取到synchronized锁，进入synchronized代码块，此时因为instance对象为null，所以，此时执行instance = new SingleInstance()语句。

（3）在执行instance = new SingleInstance()语句时，线程A会在JVM中开辟一块空白的内存空间。

（4）线程A将instance引用指向空白的内存空间，在没有进行对象初始化的时候，发生了线程切换，线程A释放synchronized锁，CPU切换到线程B上。

（5）线程B进入synchronized代码块，读取到线程A返回的instance对象，此时这个instance不为null，但是并未进行对象的初始化操作，是一个空对象。此时，线程B如果使用instance，就可能出现问题！！！

【第二种步骤】

（1）线程A先进入if条件判断，

（2）线程A获取synchronized锁，并进行第二次if条件判断，此时的instance为null，执行instance = new SingleInstance()语句。

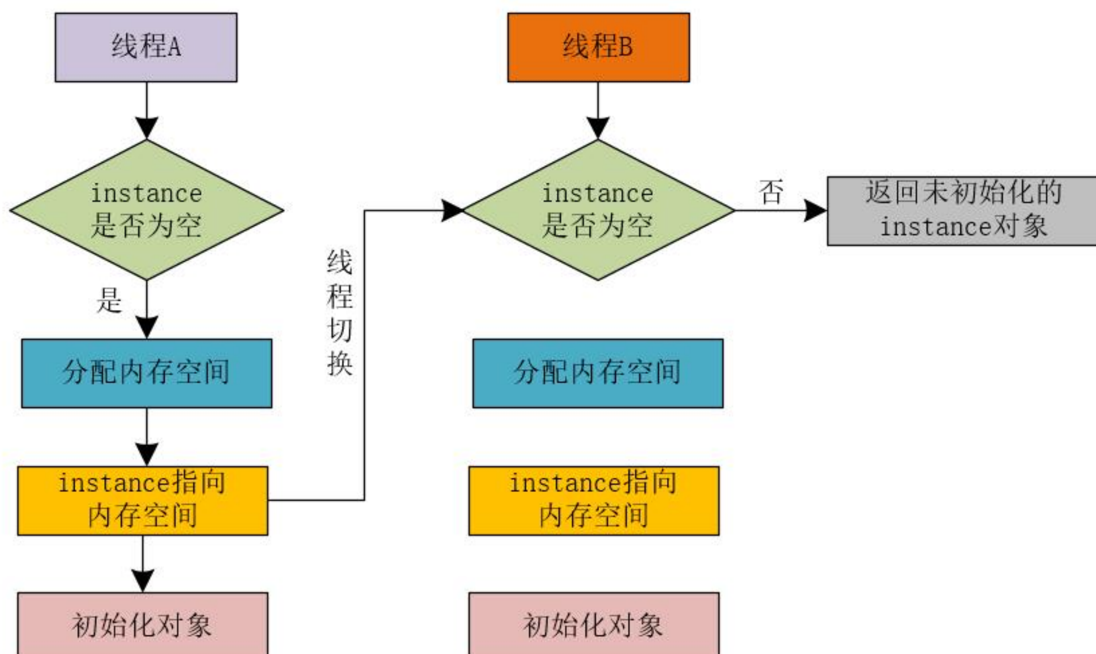
（3）线程A在JVM中开辟一块空白的内存空间。

（4）线程A将instance引用指向空白的内存空间，在没有进行对象初始化的时候，发生了线程切换，CPU切换到线程B上。

（5）线程B进行第一次if判断，发现instance对象不为null，但是此时的instance对象并未进行初始化操作，是一个空对象。如果线程B直接使用这个instance对象，就可能出现问题！！！

在第二种步骤中，即使发生线程切换时，线程A没有释放锁，则线程B进行第一次if判断时，发现instance已经不为null，直接返回instance，而无需尝试获取synchronized锁。

我们可以将上述过程简化成下图所示。



总结

导致并发编程产生各种诡异问题的根源有三个：缓存导致的可见性问题、线程切换导致的原子性问题和编译优化带来的有序性问题。我们从根源上理解了这三个问题产生的原因，能够帮助我们更好的编写高并发程序。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

如何解决可见性和有序性问题？这次彻底懂了！

写在前面

> 大冰：小菜童鞋，目前，我们把所有可见性问题、原子性问题和有序性问题都介绍完了，感觉自己有啥进步吗？
>
> 小菜：大冰哥，通过前面的学习，感觉自己进步确实挺大的，原来学习并发编程包含的知识点这么多，我之前以为只是简单的创建一个线程而已，怪不得上次我没有通过面试呢！
>
> 大冰：是的，并发编程包含的知识点很多，我们慢慢学习。之前，我们介绍了可见性问题、原子性问题和有序性问题，那么今天，我们就来讲讲如何解决可见性和有序性问题。

问题排查

我们之前通过：

《【高并发】一文解密诡异并发问题的第一个幕后黑手——可见性问题》

《【高并发】解密导致并发问题的第二个幕后黑手——原子性问题（文末有福利）》

《【高并发】解密导致并发问题的第三个幕后黑手——有序性问题》

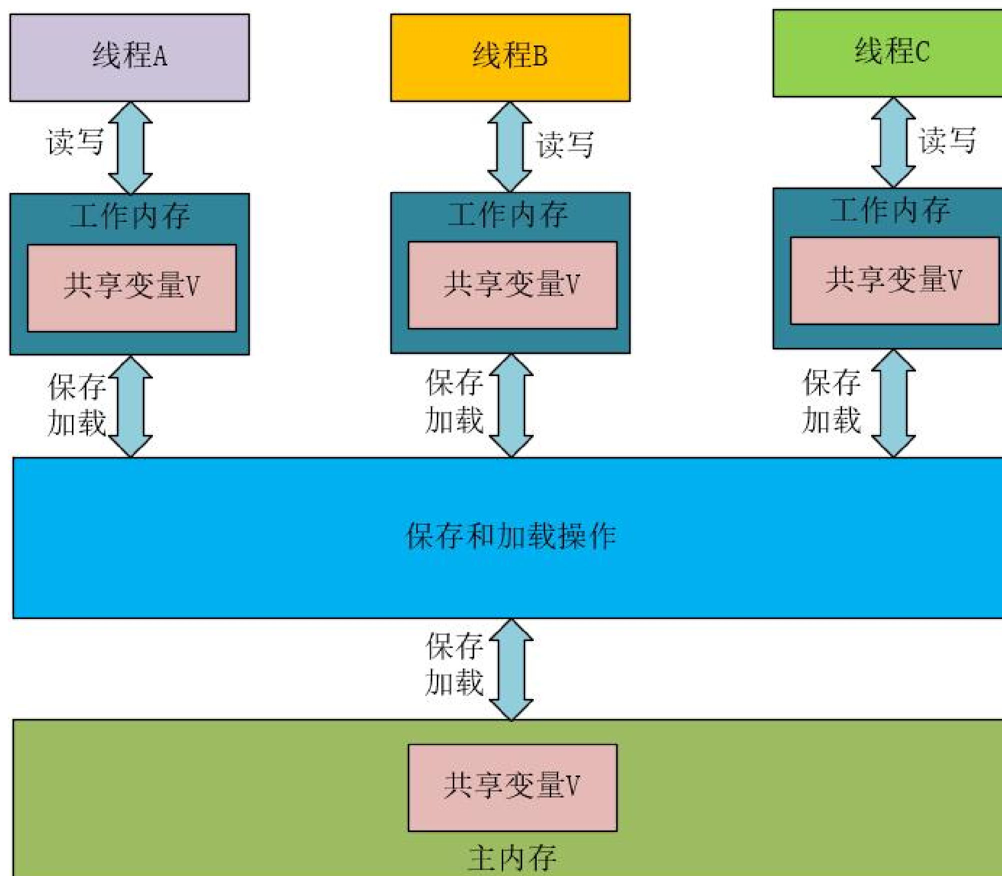
详细介绍了导致并发编程出现各种诡异问题的三个"幕后黑手"，接下来，我们就开始手撕这三个"幕后黑手"，让并发编程不再困难！

今天，我们先来看看在Java中是如何解决线程的可见性和有序性问题的，说到这，就不得不提一个Java的核心技术，那就是——**Java的内存模型**。

如果编写的并发程序出现问题时，很难通过调试来解决相应的问题，此时，需要一行行的检查代码，这个时候，如果充分理解并掌握了Java的内存模型，你就能够很快分析并定位出问题所在。

什么是Java内存模型?

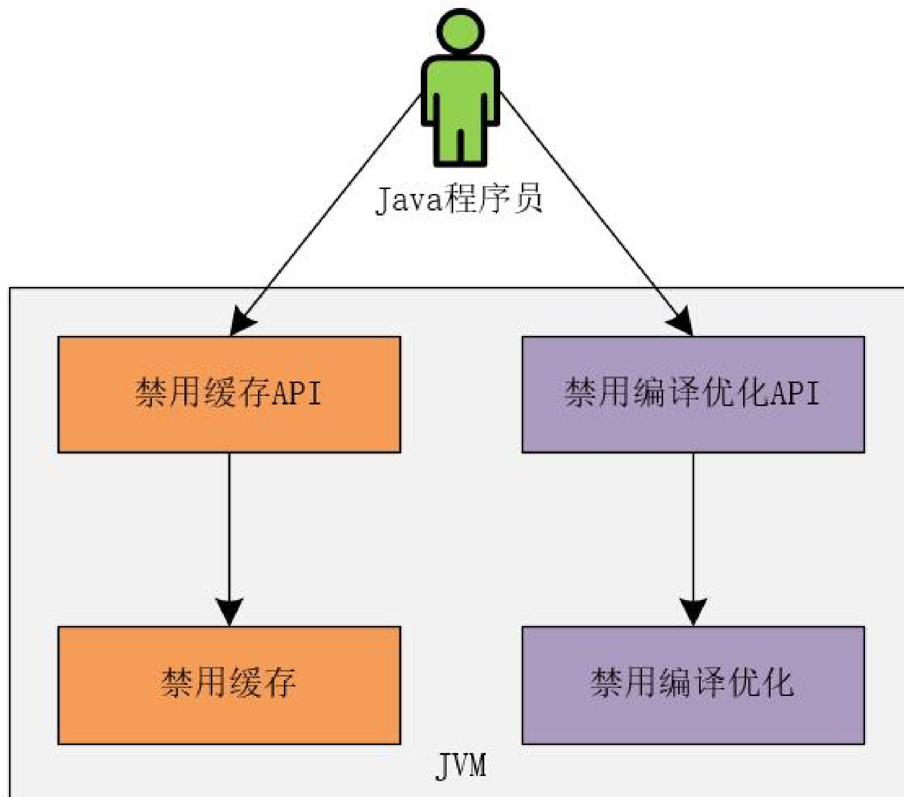
在内存里，Java内存模型规定了所有的变量都存储在主内存（物理内存）中，每条线程还有自己的工作内存，线程对变量的所有操作都必须在工作内存中进行。不同的线程无法访问其他线程的工作内存里的内容。我们可以使用下图来表示在逻辑上 **线程、主内存、工作内存**的三者交互关系。



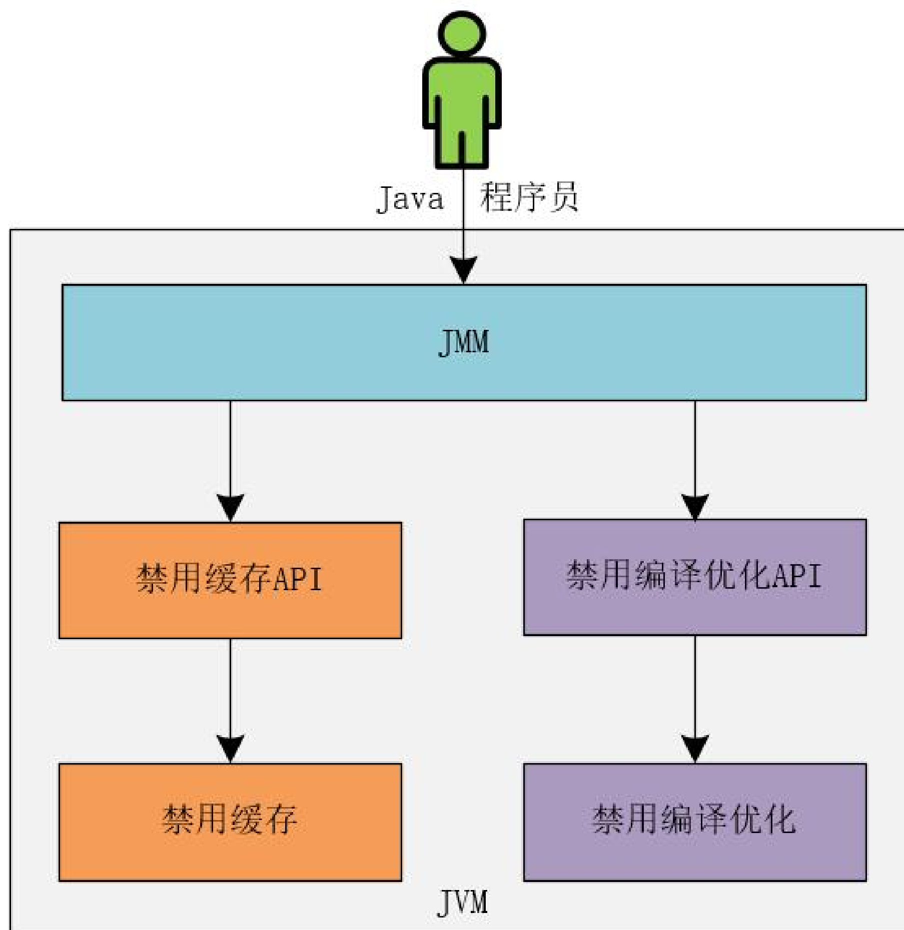
现在，我们都理解了缓存导致了可见性问题，编译优化导致了有序性问题。也就是说解决可见性和有序性问题的最直接的办法就是**禁用缓存和编译优化**。但是，如果只是简单的禁用了缓存和编译优化，那我们写的所谓的高并发程序的性能也就高不到哪去了！甚至会和单线程程序的性能没什么两样！有时，由于竞争锁的存在，可能会比单线程程序的性能还要低。

那么，既然不能完全禁用缓存和编译优化，那如何解决可见性和有序性的问题呢？其实，合理的方案应该是**按照需要禁用缓存和编译优化**。什么是按需禁用缓存和编译优化呢？简单点来说，就是需要禁用的时候禁用，不需要禁用的时候就不禁用。有些人可能会说，这不废话吗？其实不然，我们继续向下看。

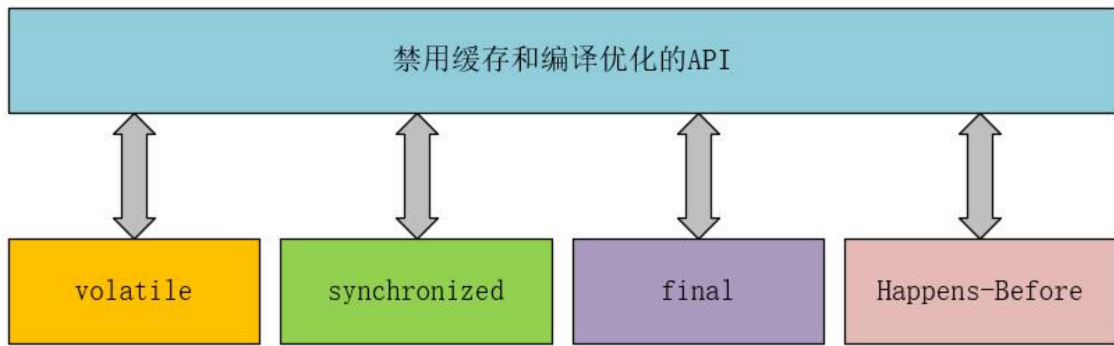
何时禁用和不禁用缓存和编译优化，可以根据编写高并发程序的开发人员的要求来合理的确定（**这里需要重点理解**）。所以，可以这么说，为了解决可见性和有序性问题，Java只需要提供给Java程序员按照需要禁用缓存和编译优化的方法即可。



Java内存模型是一个非常复杂的规范，网上关于Java内存模型的文章很多，但是大多数说的都是理论，理论说多了就成了废话。这里，我不会太多的介绍Java内存模型那些晦涩难懂的理论知识。 其实，作为开发人员，我们可以这样理解Java的内存模型：**Java内存模型规范了Java虚拟机（JVM）如何提供按需禁用缓存和编译优化的方法。**



说的具体一些，这些方法包括：volatile、synchronized和final关键字，以及Java内存模型中的Happens-Before规则。



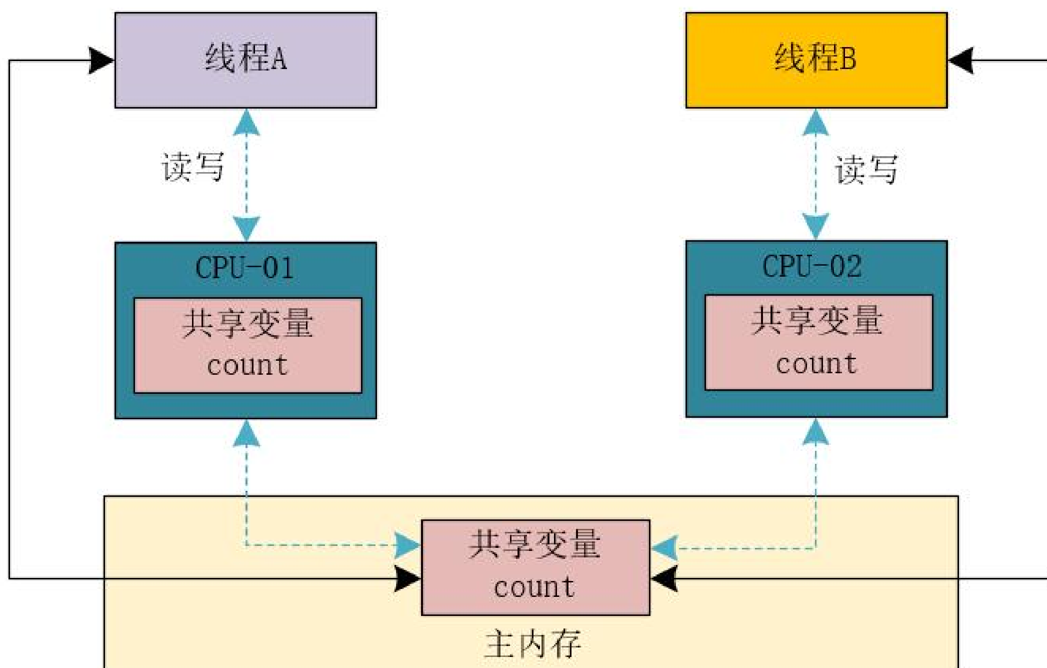
volatile为何能保证线程间可见？

volatile关键字不是Java特有的，在C语言中也存在volatile关键字，这个关键字最原始的意义就是禁用CPU缓存。

例如，我们在程序中使用volatile关键字声明了一个变量，如下所示。

```
volatile int count = 0
```

此时，Java对这个变量的读写，不能使用CPU缓存，必须从内存中读取和写入。



接下来，我们一起来看一个代码片段，如下所示。

【示例一】

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 1;
    v = true;
  }

  public void reader() {
    if (v == true) {
      //x的值是多少呢？
    }
```

```
    }
  }
```

以上示例来源于：http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#finalWrong

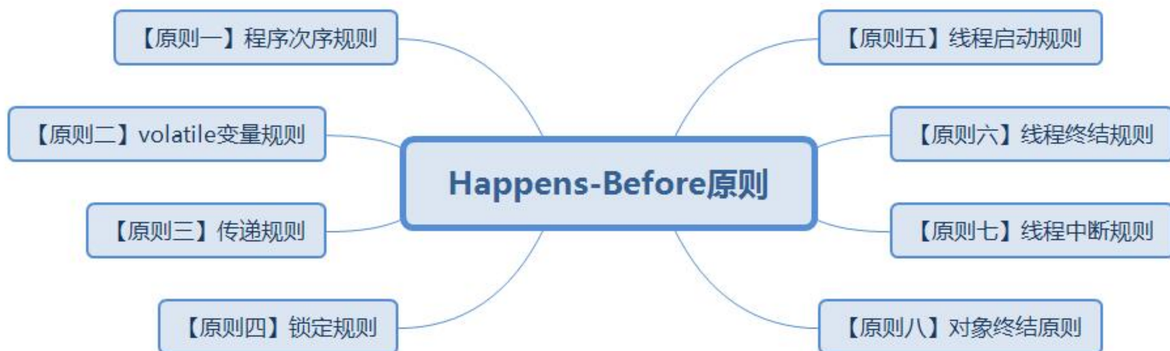
这里，假设线程A执行writer()方法，按照volatile会将v=true写入内存；线程B执行reader()方法，按照volatile，线程B会从内存中读取变量v，如果线程B读取到的变量v为true，那么，此时的变量x的值是多少呢？？

这个示例程序给人的直觉就是x的值为1，其实，x的值具体是多少和JDK的版本有关，如果使用的JDK版本低于1.5，则x的值可能为1，也可能为0。如果使用1.5及1.5以上版本的JDK，则x的值就是1。

看到这个，就会有人提出问题了？这是为什么呢？其实，答案就是在JDK1.5版本中的Java内存模型中引入了Happens-Before原则。

Happens-Before原则

我们可以将Happens-Before原则总结成如下图所示。



接下来，我们就结合案例程序来说明Java内存模型中的Happens-Before原则。

【原则一】程序次序规则

在一个线程中，按照代码的顺序，前面的操作Happens-Before于后面的任意操作。

例如【示例一】中的程序x=1会在v=true之前执行。这个规则比较符合单线程的思维：在同一个线程中，程序在前面对某个变量的修改一定是对后续操作可见的。

【原则二】volatile变量规则

对一个volatile变量的写操作，Happens-Before于后续对这个变量的读操作。

也就是说，对一个使用了volatile变量的写操作，先行发生于后面对这个变量的读操作。这个需要大家重点理解。

【原则三】传递规则

如果A Happens-Before B，并且B Happens-Before C，则A Happens-Before C。

我们结合【原则一】、【原则二】和【原则三】再来看【示例一】程序，此时，我们可以得出如下结论：

（1）x = 1 Happens-Before 写变量v = true，符合【原则一】程序次序规则。

（2）写变量v = true Happens-Before 读变量v = true，符合【原则二】volatile变量规则。

再根据【原则三】传递规则，我们可以得出结论：x = 1 Happens-Before 读变量v=true。

也就是说，如果线程B读取到了v=true，那么，线程A设置的x = 1对线程B就是可见的。换句话说，就是此时的线程B能够访问到x=1。

其实，Java 1.5版本的 java.util.concurrent并发工具就是靠volatile语义来实现可见性的。

【原则四】锁定规则

对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。

例如，下面的代码，在进入synchronized代码块之前，会自动加锁，在代码块执行完毕后，会自动释放锁。

【示例二】

```java
public class Test{
    private int x = 0;
    public void initX{
        synchronized(this){ //自动加锁
            if(this.x < 10){
                this.x = 10;
            }
        } //自动释放锁
    }
}
```

我们可以这样理解这段程序：假设变量x的值为10，线程A执行完synchronized代码块之后将x变量的值修改为10，并释放synchronized锁。当线程B进入synchronized代码块时，能够获取到线程A对x变量的写操作，也就是说，线程B访问到的x变量的值为10。

【原则五】线程启动规则

如果线程A调用线程B的start()方法来启动线程B，则start()操作Happens-Before于线程B中的任意操作。

我们也可以这样理解线程启动规则：线程A启动线程B之后，线程B能够看到线程A在启动线程B之前的操作。

我们来看下面的代码。

【示例三】

```java
//在线程A中初始化线程B
Thread threadB = new Thread(()->{
    //此处的变量x的值是多少呢？答案是100
});
//线程A在启动线程B之前将共享变量x的值修改为100
x = 100;
//启动线程B
threadB.start();
```

上述代码是在线程A中执行的一个代码片段，根据【原则五】线程的启动规则，线程A启动线程B之后，线程B能够看到线程A在启动线程B之前的操作，在线程B中访问到的x变量的值为100。

【原则六】线程终结规则

线程A等待线程B完成（在线程A中调用线程B的join()方法实现），当线程B完成后（线程A调用线程B的join()方法返回），则线程A能够访问到线程B对共享变量的操作。

例如，在线程A中进行的如下操作。

【示例四】

```java
Thread threadB = new Thread(()-{
    //在线程B中，将共享变量x的值修改为100
    x = 100;
});
//在线程A中启动线程B
threadB.start();
//在线程A中等待线程B执行完成
threadB.join();
//此处访问共享变量x的值为100
```

【原则七】线程中断规则

对线程interrupt()方法的调用Happens-Before于被中断线程的代码检测到中断事件的发生。

例如，下面的程序代码。在线程A中中断线程B之前，将共享变量x的值修改为100，则当线程B检测到中断事件时，访问到的x变量的值为100。

【示例五】

```java
    //在线程A中将x变量的值初始化为0
    private int x = 0;

    public void execute(){
        //在线程A中初始化线程B
```

```
        Thread threadB = new Thread(()->{
            //线程B检测自己是否被中断
            if (Thread.currentThread().isInterrupted()){
                //如果线程B被中断，则此时X的值为100
                System.out.println(x);
            }
        });
        //在线程A中启动线程B
        threadB.start();
        //在线程A中将共享变量X的值修改为100
        x = 100;
        //在线程A中中断线程B
        threadB.interrupt();
    }
```

【原则八】对象终结原则

一个对象的初始化完成Happens-Before于它的finalize()方法的开始。

例如，下面的程序代码。

【示例六】

```
public class TestThread {

   public TestThread(){
       System.out.println("构造方法");
   }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("对象销毁");
    }

    public static void main(String[] args){
        new TestThread();
        System.gc();
    }
}
```

运行结果如下所示。

```
构造方法
对象销毁
```

再说final关键字

使用final关键字修饰的变量，是不会被改变的。但是在Java 1.5之前的版本中，使用final修饰的变量也会出现错误的情况，在Java 1.5版本之后，Java内存模型对使用final关键字修饰的变量的重排序进行了一定的约束。只要我们能够提供正确的构造函数就不会出现问题。

例如，下面的程序代码，在构造函数中将this赋值给了全局变量global.obj，**此时对象初始化还没有完成，此时对象初始化还没有完成，此时对象初始化还没有完成**，重要的事情说三遍！！线程通过global.obj读取的x值可能为0。

【示例七】

```
final x = 0;
public FinalFieldExample() { // bad!
  x = 3;
  y = 4;
  // bad construction - allowing this to escape
  global.obj = this;
}
```

以上示例来源于：http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#finalWrong

Java内存模式的底层实现

主要是通过内存屏障(memory barrier)禁止重排序的， 即时编译器根据具体的底层体系架构， 将这些内存屏障替换成具体的 CPU 指令。 对于编译器而言，内存屏障将限制它所能做的重排序优化。 而对于处理器而言， 内存屏障将会导致缓存的刷新操作。 比如， 对于volatile， 编译器将在volatile字段的读写操作前后各插入一些内存屏障。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

synchronized原理

synchronized的基本使用

synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。synchronized的作用主要有三个：

（1）确保线程互斥的访问同步代码。

（2）保证共享变量的修改能够及时可见。

（3）有效解决重排序问题。

从语法上讲，synchronized总共有三种用法：

（1）修饰普通方法。

（2）修饰静态方法。

（3）修饰代码块。

接下来，我就通过几个例子程序来说明一下这三种使用方式（为了便于比较，三段代码除了synchronized的使用方式不同以外，其他基本保持一致）。

1、没有同步的情况

代码段一：

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public void method1(){
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public void method2(){
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
```

```
            }
        }).start();
    }
}
```

执行结果如下，线程1和线程2同时进入执行状态，线程2执行速度比线程1快，所以线程2先执行完成，这个过程中线程1和线程2是同时执行的。

```
Method 1 start
Method 1 execute
Method 2 start
Method 2 execute
Method 2 end
Method 1 end
```

2、对普通方法同步

代码段二

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public synchronized void method1(){
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public synchronized void method2(){
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
            }
        }).start();
    }
}
```

执行结果如下，跟代码段一比较，可以很明显的看出，线程2需要等待线程1的method1执行完成才能开始执行method2方法。

```
Method 1 start
Method 1 execute
Method 1 end
Method 2 start
Method 2 execute
Method 2 end
```

3、静态方法（类）同步

代码段三：

```
package com.paddx.test.concurrent;

 public class SynchronizedTest {
     public static synchronized void method1(){
         System.out.println("Method 1 start");
         try {
             System.out.println("Method 1 execute");
             Thread.sleep(3000);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
         System.out.println("Method 1 end");
     }

     public static synchronized void method2(){
         System.out.println("Method 2 start");
         try {
             System.out.println("Method 2 execute");
             Thread.sleep(1000);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
         System.out.println("Method 2 end");
     }

     public static void main(String[] args) {
         final SynchronizedTest test = new SynchronizedTest();
         final SynchronizedTest test2 = new SynchronizedTest();

         new Thread(new Runnable() {
             @Override
             public void run() {
                 test.method1();
             }
         }).start();

         new Thread(new Runnable() {
             @Override
             public void run() {
                 test2.method2();
             }
         }).start();
     }
 }
```

执行结果如下，对静态方法的同步本质上是对类的同步（静态方法本质上是属于类的方法，而不是对象上的方法），所以即使test和test2属于不同的对象，但是它们都属于SynchronizedTest类的实例，所以也只能顺序的执行method1和method2，不能并发执行。

```
Method 1 start
Method 1 execute
Method 1 end
Method 2 start
Method 2 execute
Method 2 end
```

4、代码块同步

代码段四：

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public void method1(){
        System.out.println("Method 1 start");
        try {
            synchronized (this) {
                System.out.println("Method 1 execute");
                Thread.sleep(3000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public void method2(){
        System.out.println("Method 2 start");
        try {
            synchronized (this) {
                System.out.println("Method 2 execute");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
            }
        }).start();
    }
}
```

执行结果如下，虽然线程1和线程2都进入了对应的方法开始执行，但是线程2在进入同步块之前，需要等待线程1中同步块执行完成。

```
Method 1 start
Method 1 execute
Method 2 start
Method 1 end
Method 2 execute
Method 2 end
```

synchronized 原理

如果对上面的执行结果还有疑问，也先不用急，我们先来了解synchronized的原理，再回头上面的问题就一目了然了。我们先通过反编译下面的代码来看看synchronized是如何实现对代码块进行同步的：

```java
package com.paddx.test.concurrent;

public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
            System.out.println("Method 1 start");
        }
    }
}
```

```
liuxpdeMacBook-Pro:classes liuxp$ javap -c com.paddx.test.concurrent.SynchronizedDemo
Compiled from "SynchronizedDemo.java"
public class com.paddx.test.concurrent.SynchronizedDemo {
  public com.paddx.test.concurrent.SynchronizedDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void method();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Method 1 start
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
```

关于这两条指令的作用，我们直接参考JVM规范中描述：

monitorenter ：

```
Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The
thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref,
as follows:
• If the entry count of the monitor associated with objectref is zero, the thread enters the monitor
and sets its entry count to one. The thread is then the owner of the monitor.
• If the thread already owns the monitor associated with objectref, it reenters the monitor,
incrementing its entry count.
• If another thread already owns the monitor associated with objectref, the thread blocks until the
monitor's entry count is zero, then tries again to gain ownership
```

这段话的大概意思为：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit：

```
The thread that executes monitorexit must be the owner of the monitor associated with the instance
referenced by objectref.
The thread decrements the entry count of the monitor associated with objectref. If as a result the
value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other
threads that are blocking to enter the monitor are allowed to attempt to do so.
```

这段话的大概意思为：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过这两段描述，我们应该能很清楚的看出synchronized的实现原理，synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

我们再来看一下同步方法的反编译结果：

```
package com.paddx.test.concurrent;

public class SynchronizedMethod {
    public synchronized void method() {
        System.out.println("Hello World!");
    }
}
```

反编译结果：

```
public synchronized void method();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello World!
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 5: 0
      line 6: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  this   Lcom/paddx/test/concurrent/SynchronizedMethod;
```

从反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

运行结果解释

有了对synchronized原理的认识，再来看上面的程序就可以迎刃而解了。

1、代码段2结果

虽然method1和method2是不同的方法，但是这两个方法都进行了同步，并且是通过同一个对象去调用的，所以调用之前都需要先去竞争同一个对象上的锁（monitor），也就只能互斥的获取到锁，因此，method1和method2只能顺序的执行。

2、代码段3结果

虽然test和test2属于不同对象，但是test和test2属于同一个类的不同实例，由于method1和method2都属于静态同步方法，所以调用的时候需要获取同一个类上monitor（每个类只对应一个class对象），所以也只能顺序的执行。

3、代码段4结果

对于代码块的同步实质上需要获取synchronized关键字后面括号中对象的monitor，由于这段代码中括号的内容都是this，而method1和method2又是通过同一的对象去调用的，所以进入同步块之前需要去竞争同一个对象上的锁，因此只能顺序执行同步块。

总结

synchronized是Java并发编程中最常用的用于保证线程安全的方式，其使用相对也比较简单。但是如果能够深入了解其原理，对监视器锁等底层知识有所了解，一方面可以帮助我们正确的使用synchronized关键字，另一方面也能够帮助我们更好的理解并发编程机制，有助我们在不同的情况下选择更优的并发策略来完成任务。对平时遇到的各种并发问题，也能够从容的应对。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

为何在32位多核CPU上执行long型变量的写操作会出现诡异的Bug问题？

写在前面

> 大冰：小菜童鞋，前几天讲的知识点复习了吗？
>
> 小菜：复习了，大冰哥，我回去关注了你的公众号，收藏和转发了你的文章，看了好几遍呢！！
>
> 大冰：好的，一定要好好复习啊，今天，我们来分析一个诡异的问题：**为何在32位多核CPU上执行long型变量的写操作会出现诡异的Bug问题呢？**今天的内容很重要，它能够帮助你更加深刻的理解线程的原子性问题，一定要好好听！

诡异的问题

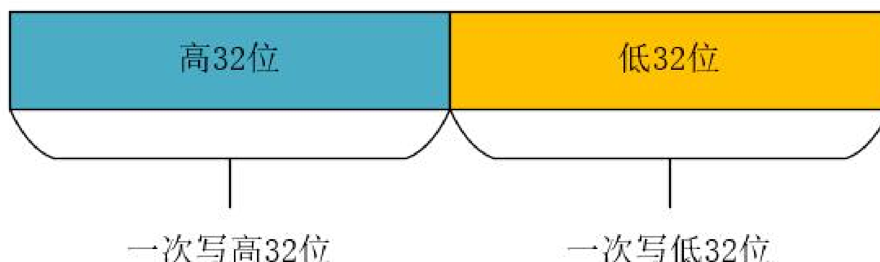
我们在32位多核CPU的计算机上以多线程的方式读写long类型的共享变量时，线程已经将变量成功写入了内存，但是重新读取出来的数据和之前写入的数据不一致，这到底是为什么呢？

原因分析

其实，造成这个问题的根本原因就是线程的原子性问题，而线程的原子性问题最终的"幕后黑手"是线程切换，如果能够禁用线程切换就能够解决这个问题了！在操作系统层面来看，操作系统做线程切换需要依赖CPU的中断机制，所以说，禁止CPU发生中断就能够禁止线程切换。

这种方案在单核CPU上是可行的，但是并不适合多核CPU。

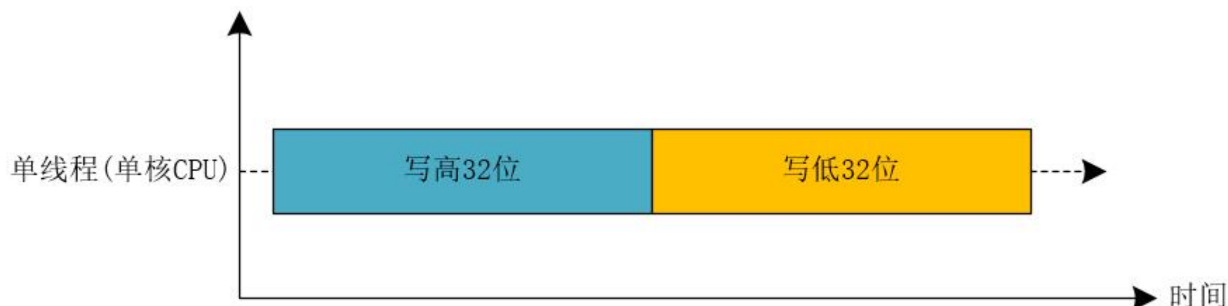
其实，就分析为何在32位多核CPU上执行long型变量的写操作会出现诡异的Bug问题，我们需要从数据类型占用的存储空间来分析。long型变量是64位的，在32位CPU上执行写操作会被拆分成两次写操作（分别是写高32位和写低32位）。我们可以用下图来表示。



32位单核CPU

在32位单核CPU场景下，同一时刻只有一个线程执行，禁止CPU中断，也就是说，在单核CPU上，操作系统不会重新调度线程，实际上，也就是禁止了线程切换。如果一个线程获取到CPU资源，就可以一直执行下去，直到线程结束为止。在这个线程中，对于long型变量的两次写操作，要么都被执行，要么都没有被执行，两次写操作具有原子性，不会出现写入的数据和读取的数据不一致的情况。

我们可以简单的使用下图来表示32位单核CPU写long型数据这个过程。



由上图我们可以看出，在32位单核CPU中，禁止了线程切换之后，所有的线程都是串行执行的，对于long型变量的两次写操作，要么都被执行，要么都没有被执行，两次写操作具有原子性，不会出现写入的数据和读取的数据不一致的情况。

32位多核CPU

在32位多核CPU场景下，同一时刻，可能有两个甚至更多的线程在同时执行。假设有两个线程分别是线程A和线程B，线程A执行在CPU-01上，线程B执行在CPU-02上，此时，禁用CPU中断，只能保证在每个CPU上执行的线程是连续的，并不能保证同一时刻只有一个线程执行，如果线程A和线程B同时写long型变量的高32位的话，那么，就有可能出现诡异的Bug问题，也就是说，**明明已经将变量成功写入内存了，但是重新读取出来的数据却不是自己写入的！！**

我们可以简单的使用下图来表示32位多核CPU并发写long型数据这个过程。

由上图我们可以看出，在32位多核CPU中，如果有多个线程同时对long类型的数据进行写操作，即使中断CPU操作，也只能保证在每个CPU上执行的线程是连续的，并不能保证同一时刻只有一个线程执行。如果多个线程同时写long型变量的高32位的话，那么，就有可能出现诡异的Bug问题。

总结

long型变量是64位的，在32位CPU上执行写操作，会被拆分成写高32位和写低32位两部分，如果此时有多个线程同时写long型变量的高32位的话，就有可能出现诡异的Bug问题。

注意：不只是long型变量，在32位多核CPU上并发写64位数据类型的数据，都会出现类似的诡异问题！！！

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

如何使用互斥锁解决多线程的原子性问题?

前言

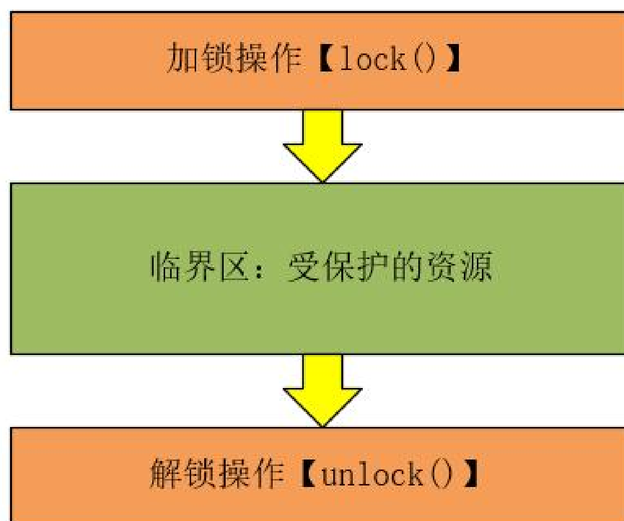
在《【高并发】如何解决可见性和有序性问题？这次彻底懂了！》一文中，我们了解了Java是如何解决多线程之间的可见性和有序性问题。另外，通过《【高并发】为何在32位多核CPU上执行long型变量的写操作会出现诡异的Bug问题？看完这篇我懂了！》一文，我们得知在32位多核CPU上读写long型数据出现问题的根本原因是**线程切换带来的原子性问题**。

如何保证原子性?

那么，如何解决线程切换带来的原子性问题呢？答案是**保证多线程之间的互斥性。也就是说，在同一时刻只有一个线程在执行！**如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，都能保证多线程之间的原子性了。

锁模型

说到线程之间的互斥，我们可以想到在并发编程中使用锁来保证线程之前的互斥性。我们可以锁模型简单的使用下图来表示。

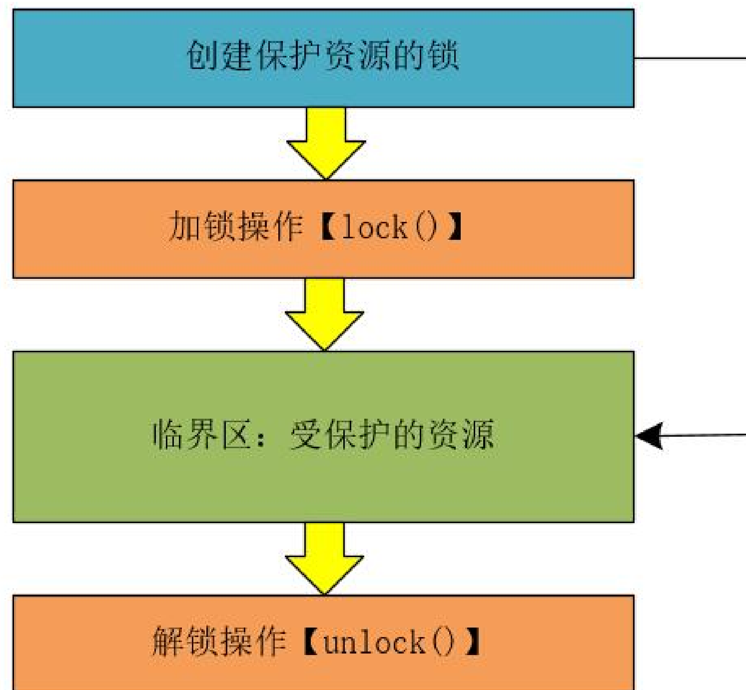


我们可以将上图中受保护的资源，也就是需要多线程之间互斥执行的代码称为临界区。线程进入临界区之前，会首先尝试加锁操作lock()，如果加锁成功，则进入临界区执行临界区中的代码，则当前线程持有锁；如果加锁失败，就会等待，直到持有锁的线程释放锁后，当前线程获取到锁进入临界区；进入临界区的线程执行完代码后，会执行解锁操作unlock()。

其实，在这个锁模型中，我们忽略了一些非常重要的内容：那就是我们对什么东西加了锁？需要我们保护的资源又是什么呢？

改进的锁模型

在并发编程中对资源进行加锁操作时，我们需要明确对什么东西加了锁？而需要我们保护的资源又是什么？只有明确了这两点，才能更好的利用Java中的互斥锁。所以，我们需要将锁模型进行修改，修改后的锁模型如下图所示。



在改进的锁模型中，首先创建一把保护资源的锁，使用这个保护资源的锁进行加锁操作，然后进入临界区执行代码，最后进行解锁操作释放锁。其中，创建的保护资源的锁，就是对临界区特定的资源进行保护。

这里需要注意的是：我们在改进的锁模型中，特意将创建保护资源的锁用箭头指向了临界区中的受保护的资源。目的是为了说明特定资源的锁是为了保护特定的资源，如果一个资源的锁保护了其他的资源，那么就会出现诡异的Bug问题，这样的Bug非常不好调试，因为我们自身会觉得，我明明已经对代码进行了加锁操作，可为什么还会出现问题呢？如果出现了这种问题，你就要排查下你创建的锁，是不是真正要保护你需要保护的资源了。

Java中的synchronized锁

说起，Java中的synchronized锁，相信大家并不陌生了，synchronized关键字可以用来修饰方法，也可以用来修饰代码块。例如，下面的代码片段所示。

```
public class LockTest{
    //创建需要加锁的对象
    private Object obj = new Object();
    //修饰代码块
    public void run(){
        synchronized(obj){
            //临界区：受保护的资源
            System.out.println("测试run()方法的同步");
        }
    }
    //使用synchronized修饰非静态方法
    public synchronized void execute(){
        //临界区：受保护的资源
        System.out.println("测试execute()方法的同步");
    }

    //使用synchronized修饰静态方法
    public synchronized static void submit(){
        //临界区：受保护的资源
        System.out.println("测试submit方法的同步");
    }
}
```

在上述的代码中，我们只是对方法（包括静态方法和非静态方法）和代码块使用了synchronized关键字，并没有执行lock()和unlock()操作。本质上，synchronized的加锁和解锁操作都是由JVM来完成的，Java编译器会在synchronized修饰的方法或代码块的前面自动加上加锁操作，而在其后面自动加上解锁操作。

在使用synchronized关键字加锁时，Java规定了一些隐式的加锁规则。

- 当使用synchronized关键字修饰代码块时，锁定的是实际传入的对象。
- 当使用synchronized关键字修饰非静态方法时，锁定的是当前实例对象this。
- 当使用synchronized关键字修饰静态方法时，锁定的是当前类的Class对象。

synchronized揭秘

使用synchronized修饰代码块和方法时JVM底层实现的JVM指令有所区别，我们以LockTest类为例，对LockTest类进行反编译，如下所示。

```
D:\>javap -c LockTest.class
Compiled from "LockTest.java"
public class io.mykit.concurrent.lab03.LockTest {
  public io.mykit.concurrent.lab03.LockTest();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: new           #2                  // class java/lang/Object
       8: dup
       9: invokespecial #1                  // Method java/lang/Object."<init>":()V
      12: putfield      #3                  // Field obj:Ljava/lang/Object;
      15: return

  public void run();
    Code:
       0: aload_0
       1: getfield      #3                  // Field obj:Ljava/lang/Object;
       4: dup
       5: astore_1
       6: monitorenter
       7: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      10: ldc           #5                  // String 测试run()方法的同步
      12: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      15: aload_1
      16: monitorexit
      17: goto          25
      20: astore_2
      21: aload_1
      22: monitorexit
      23: aload_2
      24: athrow
      25: return
    Exception table:
       from    to  target type
           7    17    20   any
          20    23    20   any

  public synchronized void execute();
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #7                  // String 测试execute()方法的同步
       5: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  public static synchronized void submit();
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #8                  // String 测试submit方法的同步
       5: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

分析反编译代码块

从反编译的结果来看，synchronized在run()方法中修饰代码块时，使用了monitorenter 和monitorexit两条指令，如下所示。

```
public void run();
  Code:
     0: aload_0
     1: getfield      #3                  // Field obj:Ljava/lang/Object;
     4: dup
     5: astore_1
     6: monitorenter
     7: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
    10: ldc           #5                  // String 测试run()方法的同步
    12: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    15: aload_1
    16: monitorexit
    17: goto          25
    20: astore_2
    21: aload_1
    22: monitorexit
    23: aload_2
    24: athrow
    25: return
  Exception table:
     from    to  target type
         7    17    20   any
        20    23    20   any
```

对于monitorenter指令，查看JVM的技术规范后，可以得知：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

对于monitorexit指令，JVM技术规范如下：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过这两段描述，我们应该能很清楚的看出synchronized的实现原理，synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

分析反编译方法

从反编译的代码来看，synchronized无论是修饰非静态方法还是修饰静态方法，其执行的流程都是一样，例如，我们这里对非静态方法execute()和静态方法submit()的反编译结果如下所示。

```
public synchronized void execute();
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #7                  // String 测试execute()方法的同步
     5: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
     8: return

public static synchronized void submit();
  Code:
     0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: ldc           #8                  // String 测试submit方法的同步
     5: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
     8: return
```

注意：我这里使用的JDK版本为1.8，其他版本的JDK可能结果不同。

再次深究count+=1的问题

如果多个线程并发的对共享变量count执行加1操作，就会出现问题。此时，我们可以使用synchronized锁来尝试解决下这个问题。

例如，TestCount类中有两个方法，一个是getCount()方法，用来获取count的值；另一个是incrementCount()方法，用来给count值加1，并且incrementCount()方法使用synchronized关键字修饰，如下所示。

```
public class TestCount{
    private long count = 0L;
    public long getCount(){
        return count;
    }
    public synchronized void incrementCount(){
        count += 1;
    }
}
```

通过上面的代码，我们肯定的是incrementCount()方法被synchronized关键字修饰后，无论是单核CPU还是多核CPU，此时只有一个线程能够执行incrementCount()方法，所以，incrementCount()方法一定可以保证原子性。

这里，我们还要思考另一个问题：上面的代码是否存在可见性问题呢？ 回答这个问题之间，我们还需要看下《【高并发】如何解决可见性和有序性问题？这次彻底懂了！》一文中，Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。**

在上面的代码中，使用synchronized关键字修饰的incrementCount()方法是互斥的，也就是说，在同一时刻只有一个线程执行incrementCount()方法中的代码；而Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。** 指的是前一个线程的解锁操作对后一个线程的加锁操作可见，再综合Happens-Before原则的【原则三】传递规则：**如果A Happens-Before B，并且B Happens-Before C，则A Happens-Before C。** 我们可以得出一个结论：**前一个线程在临界区修改的共享变量（该操作在解锁之前），对后面进入这个临界区（该操作在加锁之后）的线程是可见的。**

经过上面的分析，如果多个线程同时执行incrementCount()方法，是可以保证可见性的，也就是说，如果有100个线程同时执行incrementCount()方法，count变量的最终结果为100。

但是，还没完，TestCount类中还有一个getCount()方法，如果执行了incrementCount()方法，count变量的值对getCount()方法是可见的吗？

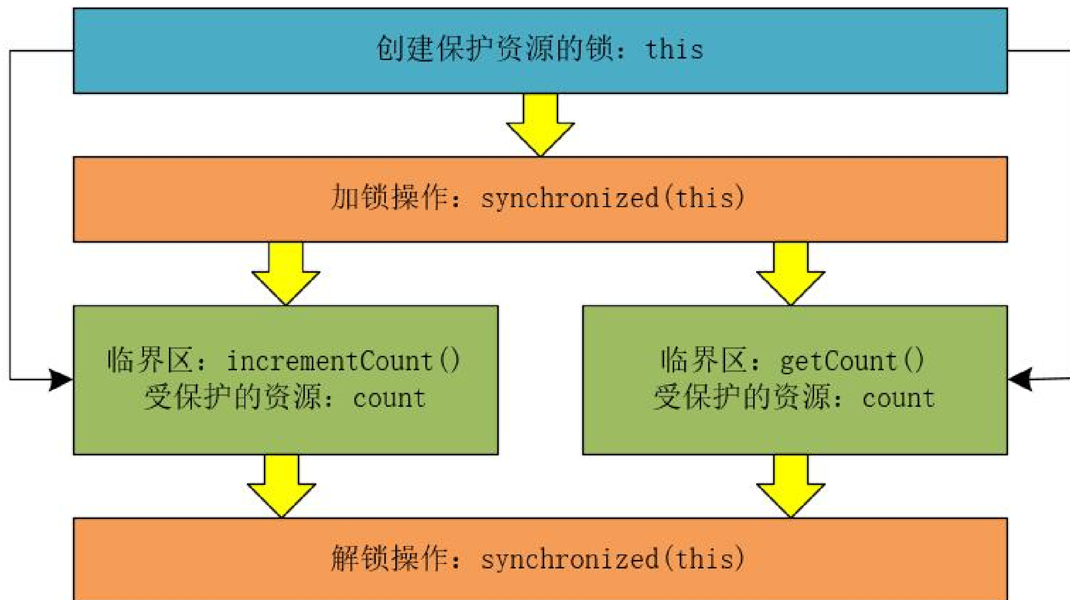
在《【高并发】如何解决可见性和有序性问题？这次彻底懂了！》一文中，Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。** 只能保证后续对这个锁的加锁的可见性。而getCount()方法没有执行加锁操作，所以，无法保证incrementCount()方法的执行结果对getCount()方法可见。

如果需要保证incrementCount()方法的执行结果对getCount()方法可见，我们也需要为getCount()方法使用synchronized关键字修饰。所以，TestCount类的代码如下所示。

```
public class TestCount{
    private long count = 0L;
    public synchronized long getCount(){
        return count;
    }
    public synchronized void incrementCount(){
        count += 1;
    }
}
```

此时，为getCount()方法也添加了synchronized锁，而且getCount()方法和incrementCount()方法锁定的都是this对象，线程进入getCount()方法和incrementCount()方法时，必须先获得this这把锁，所以，getCount()方法和incrementCount()方法是互斥的。也就是说，此时，incrementCount()方法的执行结果对getCount()方法可见。

我们也可以简单的使用下图来表示这个互斥的逻辑。



修改测试用例

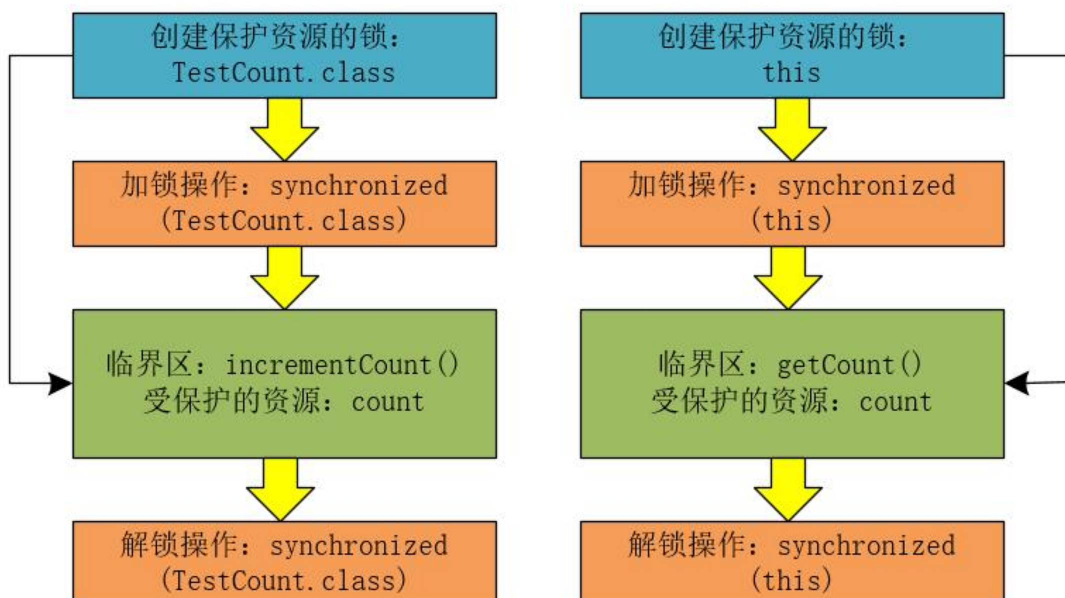
我们将上面的测试代码稍作修改，将count的修改为静态变量，将incrementCount()方法修改为静态方法。此时的代码如下所示。

```
public class TestCount{
    private static long count = 0L;
    public synchronized long getCount(){
        return count;
    }
    public synchronized static void incrementCount(){
        count += 1;
    }
}
```

那么，问题来了，getCount()方法和incrementCount()方法是否存在并发问题呢？

接下来，我们一起分析下这段代码：其实这段代码中是在用两个不同的锁来保护同一个资源count，两个锁分别为this对象和TestCount.class对象。也就是说，getCount()方法和incrementCount()方法获取的是两个不同的锁，二者的临界区没有互斥关系，incrementCount()方法对count变量的修改无法保证对getCount()方法的可见性。所以，**修改后的代码会存在并发问题。**

我们也可以使用下图来简单的表示这个逻辑。



总结

保证多线程之间的互斥性。也就是说，在同一时刻只有一个线程在执行！如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，都能保证多线程之间的原子性了。

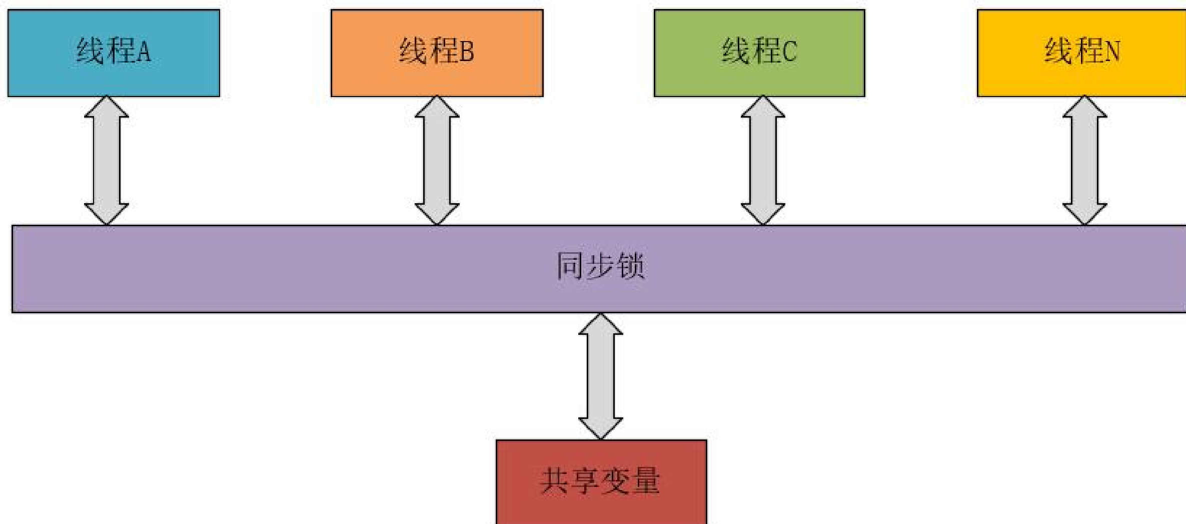
注意：在Java中，也可以使用Lock锁来实现多线程之间的互斥，大家可以自行使用Lock锁实现。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

ThreadLocal学会了这些，你也能和面试官扯皮了！

前言

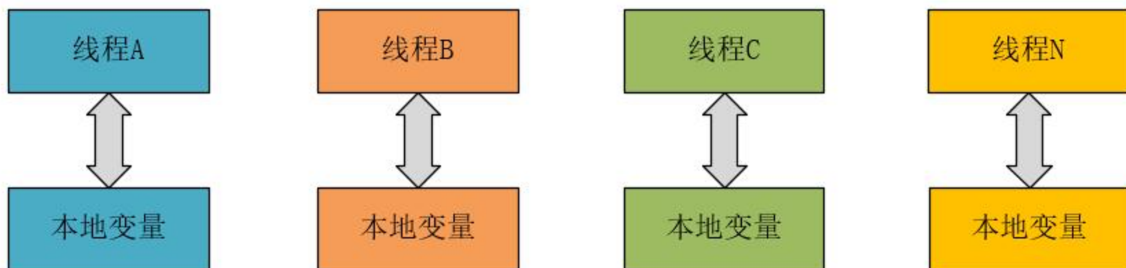
我们都知道，在多线程环境下访问同一个共享变量，可能会出现线程安全的问题，为了保证线程安全，我们往往会在访问这个共享变量的时候加锁，以达到同步的效果，如下图所示。



对共享变量加锁虽然能够保证线程的安全，但是却增加了开发人员对锁的使用技能，如果锁使用不当，则会导致死锁的问题。而**ThreadLocal能够做到在创建变量后，每个线程对变量访问时访问的是线程自己的本地变量。**

什么是ThreadLocal?

> ThreadLocal是JDK提供的，支持线程本地变量。也就是说，如果我们创建了一个ThreadLocal变量，则访问这个变量的每个线程都会有这个变量的一个本地副本。如果多个线程同时对这个变量进行读写操作时，实际上操作的是线程自己本地内存中的变量，从而避免了线程安全的问题。



ThreadLocal使用示例

例如，我们使用ThreadLocal保存并打印相关的变量信息，程序如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA: " + Thread.currentThread().getName());
```

```
            System.out.println("线程A本地变量中的值为: " + threadLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB: " + Thread.currentThread().getName());
            System.out.println("线程B本地变量中的值为: " + threadLocal.get());
        });
        //启动线程A和线程B
        threadA.start();
        threadB.start();
    }
}
```

运行程序，打印的结果信息如下所示。

```
线程A本地变量中的值为: ThreadA: Thread-0
线程B本地变量中的值为: ThreadB: Thread-1
```

此时，我们为线程A增加删除ThreadLocal中的变量的操作，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA: " + Thread.currentThread().getName());
            System.out.println("线程A本地变量中的值为: " + threadLocal.get());
            threadLocal.remove();
            System.out.println("线程A删除本地变量后ThreadLocal中的值为: " + threadLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB: " + Thread.currentThread().getName());
            System.out.println("线程B本地变量中的值为: " + threadLocal.get());
            System.out.println("线程B没有删除本地变量: " + threadLocal.get());
        });
        //启动线程A和线程B
        threadA.start();
        threadB.start();
    }
}
```

此时的运行结果如下所示。

```
线程A本地变量中的值为: ThreadA: Thread-0
线程B本地变量中的值为: ThreadB: Thread-1
线程B没有删除本地变量: ThreadB: Thread-1
线程A删除本地变量后ThreadLocal中的值为: null
```

通过上述程序我们可以看出，**线程A和线程B存储在ThreadLocal中的变量互不干扰，线程A存储的变量只能由线程A访问，线程B存储的变量只能由线程B访问。**

ThreadLocal原理

首先，我们看下Thread类的源码，如下所示。

```
public class Thread implements Runnable {
    /***********省略N行代码*************/
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
     /***********省略N行代码*************/
}
```

由Thread类的源码可以看出，在ThreadLocal类中存在成员变量threadLocals和inheritableThreadLocals，这两个成员变量都是ThreadLocalMap类型的变量，而且二者的初始值都为null。只有当前线程第一次调用ThreadLocal的set()方法或者get()方法时才会实例化变量。

这里需要注意的是：**每个线程的本地变量不是存放在ThreadLocal实例里面的，而是存放在调用线程的threadLocals变量里面的。** 也就是说，调用ThreadLocal的set()方法存储的本地变量是存放在具体线程的内存空间中的，而ThreadLocal类只是提供了set()和get()方法来存储和读取本地变量的值，当调用ThreadLocal类的set()方法时，把要存储的值放入**调用线程**的threadLocals中存储起来，当调用ThreadLocal类的get()方法时，从当前线程的threadLocals变量中将存储的值取出来。

接下来，我们分析下ThreadLocal类的set()、get()和remove()方法的实现逻辑。

set()方法

set()方法的源代码如下所示。

```
public void set(T value) {
    //获取当前线程
    Thread t = Thread.currentThread();
    //以当前线程为Key，获取ThreadLocalMap对象
    ThreadLocalMap map = getMap(t);
    //获取的ThreadLocalMap对象不为空
    if (map != null)
        //设置value的值
        map.set(this, value);
    else
        //获取的ThreadLocalMap对象为空，创建Thread类中的threadLocals变量
        createMap(t, value);
}
```

在set()方法中，首先获取调用set()方法的线程，接下来，使用当前线程作为Key调用getMap(t)方法来获取ThreadLocalMap对象，getMap(Thread t)的方法源码如下所示。

```
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

可以看到，getMap(Thread t)方法获取的是线程变量自身的threadLocals成员变量。

在set()方法中，如果调用getMap(t)方法返回的对象不为空，则把value值设置到Thread类的threadLocals成员变量中，而传递的key为当前ThreadLocal的this对象，value就是通过set()方法传递的值。

如果调用getMap(t)方法返回的对象为空，则程序调用createMap(t, value)方法来实例化Thread类的threadLocals成员变量。

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

也就是创建当前线程的threadLocals变量。

get()方法

get()方法的源代码如下所示。

```
public T get() {
    //获取当前线程
    Thread t = Thread.currentThread();
    //获取当前线程的threadLocals成员变量
    ThreadLocalMap map = getMap(t);
    //获取的threadLocals变量不为空
    if (map != null) {
        //返回本地变量对应的值
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //初始化threadLocals成员变量的值
    return setInitialValue();
}
```

通过当前线程来获取threadLocals成员变量，如果threadLocals成员变量不为空，则直接返回当前线程绑定的本地变量，否则调用setInitialValue()方法初始化threadLocals成员变量的值。

```
private T setInitialValue() {
    //调用初始化value的方法
    T value = initialValue();
    Thread t = Thread.currentThread();
    //根据当前线程获取threadLocals成员变量
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //threadLocals不为空，则设置value值
        map.set(this, value);
    else
        //threadLocals为空,创建threadLocals变量
        createMap(t, value);
    return value;
}
```

其中，initialValue()方法的源码如下所示。

```
protected T initialValue() {
    return null;
}
```

通过initialValue()方法的源码可以看出，这个方法可以由子类覆写，在ThreadLocal类中，这个方法直接返回null。

remove()方法

remove()方法的源代码如下所示。

```java
public void remove() {
    //根据当前线程获取threadLocals成员变量
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        //threadLocals成员变量不为空，则移除value值
        m.remove(this);
}
```

remove()方法的实现比较简单，首先根据当前线程获取threadLocals成员变量，不为空，则直接移除value的值。

注意：如果调用线程一致不终止，则本地变量会一直存放在调用线程的threadLocals成员变量中，所以，如果不需要使用本地变量时，可以通过调用ThreadLocal的remove()方法，将本地变量从当前线程的threadLocals成员变量中删除，以免出现内存溢出的问题。

ThreadLocal变量不具有传递性

> 使用ThreadLocal存储本地变量不具有传递性，也就是说，同一个ThreadLocal在父线程中设置值后，在子线程中是无法获取到这个值的，这个现象说明ThreadLocal中存储的本地变量不具有传递性。

接下来，我们来看一段代码，如下所示。

```java
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //在主线程中设置值
        threadLocal.set("ThreadLocalTest");
        //在子线程中获取值
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("子线程获取值：" + threadLocal.get());
            }
        });
        //启动子线程
        thread.start();
        //在主线程中获取值
        System.out.println("主线程获取值：" + threadLocal.get());
    }
}
```

运行这段代码输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：null
```

通过上述程序，我们可以看出在主线程中向ThreadLocal设置值后，在子线程中是无法获取到这个值的。**那有没有办法在子线程中获取到主线程设置的值呢？此时，我们可以使用InheritableThreadLocal来解决这个问题。**

InheritableThreadLocal使用示例

InheritableThreadLocal类继承自ThreadLocal类，它能够让子线程访问到在父线程中设置的本地变量的值，例如，我们将ThreadLocalTest类中的threadLocal静态变量改写成InheritableThreadLocal类的实例，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new InheritableThreadLocal<String>();

    public static void main(String[] args){
        //在主线程中设置值
        threadLocal.set("ThreadLocalTest");
        //在子线程中获取值
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("子线程获取值：" + threadLocal.get());
            }
        });
        //启动子线程
        thread.start();
        //在主线程中获取值
        System.out.println("主线程获取值：" + threadLocal.get());
    }
}
```

此时，运行程序输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：ThreadLocalTest
```

可以看到，**使用InheritableThreadLocal类存储本地变量时，子线程能够获取到父线程中设置的本地变量。**

双击评论 666

InheritableThreadLocal原理

首先，我们来看下InheritableThreadLocal类的源码，如下所示。

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
    protected T childValue(T parentValue) {
        return parentValue;
    }

    ThreadLocalMap getMap(Thread t) {
       return t.inheritableThreadLocals;
    }

    void createMap(Thread t, T firstValue) {
        t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);
    }
}
```

由InheritableThreadLocal类的源代码可知，InheritableThreadLocal类继承自ThreadLocal类，并且重写了ThreadLocal类的childValue()方法、getMap()方法和createMap()方法。也就是说，当调用ThreadLocal的set()方法时，创建的是当前Thread线程的inheritableThreadLocals成员变量而不再是threadLocals成员变量。

这里，**我们需要思考一个问题：InheritableThreadLocal类的childValue()方法是何时被调用的呢？**这就需要我们来看下Thread类的构造方法了，如下所示。

```
public Thread() {
     init(null, null, "Thread-" + nextThreadNum(), 0);
 }

public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}

Thread(Runnable target, AccessControlContext acc) {
    init(null, target, "Thread-" + nextThreadNum(), 0, acc, false);
}

public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum(), 0);
}

public Thread(String name) {
    init(null, null, name, 0);
}

public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}

public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    init(group, target, name, stackSize);
}
```

可以看到，Thread类的构造方法最终调用的是init()方法，那我们就来看下init()方法，如下所示。

```
private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc,
                      boolean inheritThreadLocals) {
       /************省略部分源码************/
        if (inheritThreadLocals && parent.inheritableThreadLocals != null)
            this.inheritableThreadLocals =
                ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
        /* Stash the specified stack size in case the VM cares */
        this.stackSize = stackSize;

        /* Set thread ID */
        tid = nextThreadID();
    }
```

可以看到，在init()方法中会判断传递的inheritThreadLocals变量是否为true，同时父线程中的inheritableThreadLocals是否为null，如果传递的inheritThreadLocals变量为true，同时，父线程中的inheritableThreadLocals不为null，则调用ThreadLocal类的createInheritedMap()方法。

```
static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

在createInheritedMap()中，使用父线程的inheritableThreadLocals变量作为参数创建新的ThreadLocalMap对象。然后在Thread类的init()方法中会将这个ThreadLocalMap对象赋值给子线程的inheritableThreadLocals成员变量。

接下来，我们来看看ThreadLocalMap的构造函数都干了啥，如下所示。

```
private ThreadLocalMap(ThreadLocalMap parentMap) {
    Entry[] parentTable = parentMap.table;
    int len = parentTable.length;
    setThreshold(len);
    table = new Entry[len];

    for (int j = 0; j < len; j++) {
        Entry e = parentTable[j];
        if (e != null) {
            @SuppressWarnings("unchecked")
            ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
            if (key != null) {
                //调用重写的childValue方法
                Object value = key.childValue(e.value);
                Entry c = new Entry(key, value);
                int h = key.threadLocalHashCode & (len - 1);
                while (table[h] != null)
                    h = nextIndex(h, len);
                table[h] = c;
                size++;
            }
        }
    }
}
```

在ThreadLocalMap的构造函数中，调用了InheritableThreadLocal类重写的childValue()方法。而InheritableThreadLocal类通过重写getMap()方法和createMap()方法，让本地变量保存到了Thread线程的inheritableThreadLocals变量中，线程通过InheritableThreadLocal类的set()方法和get()方法设置变量时，就会创建当前线程的inheritableThreadLocals变量。此时，如果父线程创建子线程，在Thread类的构造函数中会把父线程中的inheritableThreadLocals变量里面的本地变量复制一份保存到子线程的inheritableThreadLocals变量中。

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

学好并发编程，关键是要理解这三个核心问题

写在前面

写【高并发专题】有一段时间了，一些读者朋友留言说，并发编程很难，学习了很多的知识，但是在实际工作中却无从下手。对于一个线上产生的并发问题，又不知产生这个问题的原因究竟是什么。对于并发编程，感觉上似乎是掌握了，但是真正用起来却不是那么回事！

其实，造成这种现象的本质原因就是没有透彻的理解并发编程的精髓，而学好并发编程的关键是需要弄懂三个核心问题：**分工、同步和互斥。**

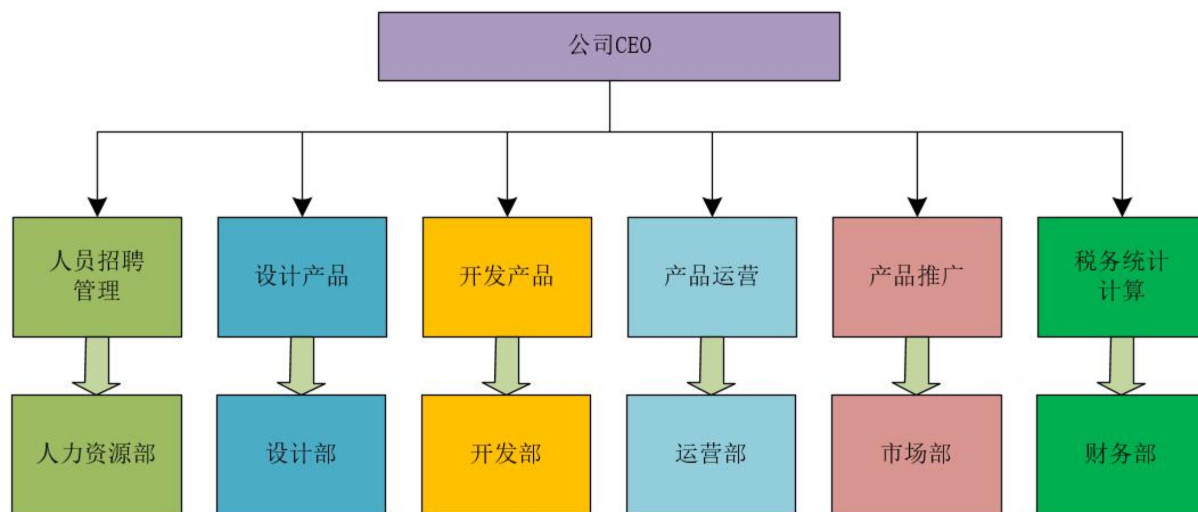
分工

> 比较官方的解释为：分工就是将一个比较大的任务，拆分成多个大小合适的任务，交给合适的线程去完成，强调的是性能。

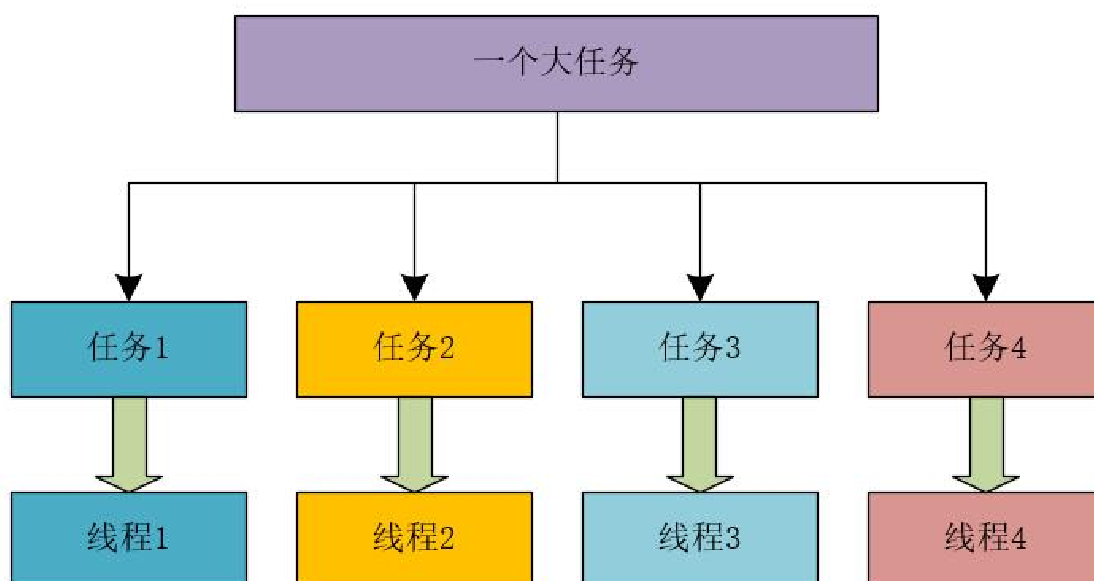
如果你还不能够理解什么是分工，这里，我们可以做一个假设。假设你是一个XXX上市公司的CEO，你的工作是如何管理好你的公司。但是，就如何管理好公司而言，涉及到的任务就比较多了，我们可以将其看做一个很大的任务，这个很大的任务，细看的话可以包括：人员招聘和管理、产品设计和开发、运营和推广、公司税务等等。那细化后这么多的任务交给你一个人去做，想必你一定是崩溃的。即使你能够挺住，估计你一个人把这所有的任务完成，那黄花菜也就凉了！到时，估计你就会偷偷的躲在角落里唱"凉凉"了。。。

所以，如果你真的想管理好你的公司，你就需要将这些任务分解，分工细化，将人员招聘和管理的任务交给人力资源部门去完成，将产品的设计交给设计部门去完成，将产品的开发交给开发部门去完成，将运营和推广交给运营和市场部门去完成，将公司税务交给财务部门去完成。此时，你的任务就是及时了解各个部门的工作情况，统筹并协调各部门的工作，并思考如何规划公司的未来。

其实，这里你将管理公司的任务拆解、细化分工之后，你会发现，其实各部门之间的工作是并行执行的。比如：人力资源部门在管理员工的绩效考核时，同时产品设计和开发部门正在设计和开发公司的产品，与此同时，公司的运营正在和设计与开发沟通如何更好的完善公司的产品，而推广部门正在加大力度宣传和推广公司的产品。而财务部门正在统计和计算公司的各种财务报表等。一切都是那么的有条不紊！



所以，安排合适的人去做合适的事情，在实际工作中是非常重要的。映射到并发编程领域也是同样的道理。如果将所有的任务交给一个线程执行，就好比将公司的所有事情交给你一个人去做一样。等到把事情做完了，黄花菜也凉了。所以，在并发编程中，我们同样需要将任务进行拆解，分工给合适的线程去完成。



在并发编程领域，还需要注意一个问题就是：**分工给合适的线程去做**。也就是说，**应该主线程执行的任务不要交给子线程去做，否则，是解决不了问题的**。这就好比一家公司的CEO将如何规划公司的未来交给一个产品开发人员去做一样，这不仅不能规划好公司的未来，甚至会与公司的价值观背道而驰。

在JavaSDK中的：Executor、Fork/Join和Future都是实现分工的一种方式。

同步

> 在并发编程中的同步，主要指的是一个线程执行完任务后，如何通知其他的线程继续执行，强调的是性能。

将任务拆分，并且合理的分工给了每个人，接下来就是如何同步每个人的任务了。

假设小明是一名前端开发人员，他渲染页面的数据需要等待小刚的接口完成，而小刚写接口又需要等待小李的服务开发完成。也就是说，**任务之间是存在依赖关系的，前面的任务完成后，才能进行后面的任务。**

对于实际工作中，这种任务的同步，大多数靠的是人与人之间的沟通，小李的服务写完了，告诉小刚，小刚则马上进行接口开发，等小刚的接口开发完成后，又告诉了小明，小明马上调用接口将返回的数据渲染在页面上。

这种同步机制映射到并发编程领域，就是**一个线程的任务执行完毕之后，通知其他的后续线程执行任务。**

对于这种线程之间的同步，我们可以使用下面的 **if** 伪代码来表示。

```
if(前面的任务完成){
    执行当前任务
}else{
    继续等待前面任务的执行
}
```

如果为了更能够及时的判断出前面的任务是否已经完成，我们也可以使用 **while** 伪代码来表示。

```
while(前面的任务未完成){
    继续等待前面任务的执行
}
执行当前任务
```

上述伪代码表示的意义是相同的：**当线程执行的条件不满足时，线程需要继续等待，一旦条件满足，就需要唤醒等待的线程继续执行。**

在并发编程领域，一个典型的场景就是生产者-消费者模型。当队列满时，生产者线程需要等待，队列不满时，需要唤醒生产者线程；当队列为空时，消费者线程需要等待，队列不空时，需要唤醒消费者。我们可以使用下面的伪代码来表示生产者-消费者模型。

- 生产者

```
while(队列已满){
    生产者线程等待
}
唤醒生产者
```

- 消费者

```
while(队列为空){
    消费者等待
}
唤醒消费者
```

在Java的SDK中，提供了一些实现线程之间同步的工具类，比如说：CountDownLatch、 CyclicBarrier 等。

互斥

> 同一时刻，只允许一个线程访问共享变量，强调的是线程执行任务的正确性。

在并发编程领域，分工和同步强调的是执行任务的性能，而线程之间的互斥则强调的是线程执行任务的正确性，也就是**线程的安全问题**。如果多个线程同时访问同一个共享变量，则可能会发生意想不到的后果，而这种意想不到的后果主要是由线程的**可见性、原子性和有序性**问题产生的。而解决可见性、原子性和有序性问题的核心，就是互斥。

关于互斥，我们可以用现实中的一个场景来描述：多个岔路口的车辆需要汇入一条道路中，而这条道路一次只能允许通过一辆车，此时，车辆就需要排队依次进入路口。

Java中提供的synchronized、Lock、ThreadLocal、final关键字等都可以解决互斥的问题。

例如，我们以synchronized为例来说明如何进行线程间的互斥，伪代码如下所示。

```
//修饰方法
public synchronized void xxx(){

}
//修饰代码块
```

```
public void xxx(){
    synchronized(obj){

    }
}
//修饰代码块
public void xxx(){
    synchronized(xxx.class){

    }
}
//修饰静态方法
public synchronized static void xxx(){

}
```

总结

并发编程旨在最大限度的利用计算机的资源，提高程序执行的性能，这需要线程之间的分工和同步来实现，在保证性能的同时，又需要保证线程的安全，这就又需要保证线程之间的互斥性。而并发编程的难点问题，往往又是由可见性、原子性和有序性问题导致的。所以，我们在学习并发编程时，一定要先弄懂线程之间的分工、同步和互斥。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

什么是ForkJoin？看这一篇就够了！

写在前面

> 在JDK中，提供了这样一种功能：它能够将复杂的逻辑拆分成一个个简单的逻辑来并行执行，待每个并行执行的逻辑执行完成后，再将各个结果进行汇总，得出最终的结果数据。有点像Hadoop中的MapReduce。
>
> ForkJoin是由JDK1.7之后提供的多线程并发处理框架。ForkJoin框架的基本思想是分而治之。什么是分而治之？分而治之就是将一个复杂的计算，按照设定的阈值分解成多个计算，然后将各个计算结果进行汇总。相应的，ForkJoin将复杂的计算当做一个任务，而分解的多个计算则是当做一个个子任务来并行执行。

Java并发编程的发展

对于Java语言来说，生来就支持多线程并发编程，在并发编程领域也是在不断发展的。Java在其发展过程中对并发编程的支持越来越完善也正好印证了这一点。

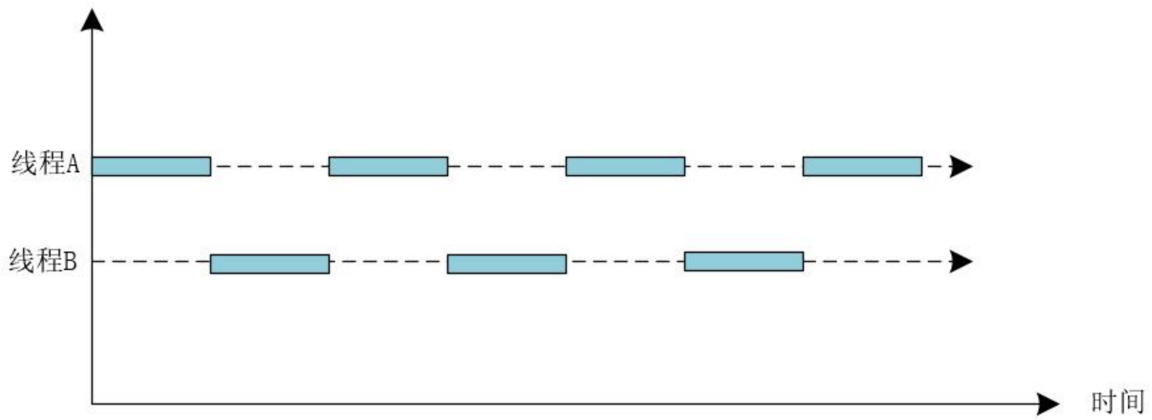
- Java 1 支持thread，synchronized。
- Java 5 引入了 thread pools， blocking queues, concurrent collections，locks, condition queues。
- Java 7 加入了fork-join库。
- Java 8 加入了 parallel streams。

并发与并行

并发和并行在本质上还是有所区别的。

并发

并发指的是在同一时刻，只有一个线程能够获取到CPU执行任务，而多个线程被快速的轮换执行，这就使得在宏观上具有多个线程同时执行的效果，并发不是真正的同时执行，并发可以使用下图表示。



并行

并行指的是无论何时，多个线程都是在多个CPU核心上同时执行的，是真正的同时执行。

分治法

基本思想

> 把一个规模大的问题划分为规模较小的子问题，然后分而治之，最后合并子问题的解得到原问题的解。

步骤

①分割原问题；

②求解子问题；

③合并子问题的解为原问题的解。

我们可以使用如下伪代码来表示这个步骤。

```
if(任务很小）{
    直接计算得到结果
}else{
    分拆成N个子任务
    调用子任务的fork()进行计算
    调用子任务的join()合并计算结果
}
```

在分治法中，子问题一般是相互独立的，因此，经常通过递归调用算法来求解子问题。

典型应用

- 二分搜索
- 大整数乘法
- Strassen矩阵乘法
- 棋盘覆盖
- 合并排序

- 快速排序
- 线性时间选择
- 汉诺塔

ForkJoin并行处理框架

ForkJoin框架概述

Java 1.7 引入了一种新的并发框架—— Fork/Join Framework，主要用于实现“分而治之”的算法，特别是分治之后递归调用的函数。

ForkJoin框架的本质是一个用于并行执行任务的框架， 能够把一个大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务的计算结果。在Java中，ForkJoin框架与ThreadPool共存，并不是要替换ThreadPool

其实，在Java 8中引入的并行流计算，内部就是采用的ForkJoinPool来实现的。例如，下面使用并行流实现打印数组元组的程序。

```
public class SumArray {
    public static void main(String[] args){
        List<Integer> numberList = Arrays.asList(1,2,3,4,5,6,7,8,9);
        numberList.parallelStream().forEach(System.out::println);
    }
}
```

这段代码的背后就使用到了ForkJoinPool。

说到这里，可能有读者会问：**可以使用线程池的ThreadPoolExecutor来实现啊？为什么要使用ForkJoinPool啊？ForkJoinPool是个什么鬼啊？！** 接下来，我们就来回答这个问题。

ForkJoin框架原理

ForkJoin框架是从jdk1.7中引入的新特性,它同ThreadPoolExecutor一样，也实现了Executor和ExecutorService接口。它使用了一个无限队列来保存需要执行的任务，而线程的数量则是通过构造函数传入，如果没有向构造函数中传入指定的线程数量，那么当前计算机可用的CPU数量会被设置为线程数量作为默认值。

ForkJoinPool主要使用**分治法(Divide-and-Conquer Algorithm)**来解决问题。典型的应用比如快速排序算法。这里的要点在于，ForkJoinPool能够使用相对较少的线程来处理大量的任务。比如要对1000万个数据进行排序，那么会将这个任务分割成两个500万的排序任务和一个针对这两组500万数据的合并任务。以此类推，对于500万的数据也会做出同样的分割处理，到最后会设置一个阈值来规定当数据规模到多少时，停止这样的分割处理。比如，当元素的数量小于10时，会停止分割，转而使用插入排序对它们进行排序。那么到最后，所有的任务加起来会有大概200万+个。问题的关键在于，对于一个任务而言，只有当它所有的子任务完成之后，它才能够被执行。

所以当使用ThreadPoolExecutor时，使用分治法会存在问题，因为ThreadPoolExecutor中的线程无法向任务队列中再添加一个任务并在等待该任务完成之后再继续执行。而使用ForkJoinPool就能够解决这个问题，它就能够让其中的线程创建新的任务，并挂起当前的任务，此时线程就能够从队列中选择子任务执行。

那么使用ThreadPoolExecutor或者ForkJoinPool，性能上会有什么差异呢？

首先，使用ForkJoinPool能够使用数量有限的线程来完成非常多的具有父子关系的任务，比如使用4个线程来完成超过200万个任务。但是，使用ThreadPoolExecutor时，是不可能完成的，因为ThreadPoolExecutor中的Thread无法选择优先执行子任务，需要完成200万个具有父子关系的任务时，也需要200万个线程，很显然这是不可行的，也是很不合理的！！

工作窃取算法

假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点：
充分利用线程进行并行计算，并减少了线程间的竞争。

工作窃取算法的缺点：
在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且该算法会消耗更多的系统资源，比如创建多个线程和多个双端队列。

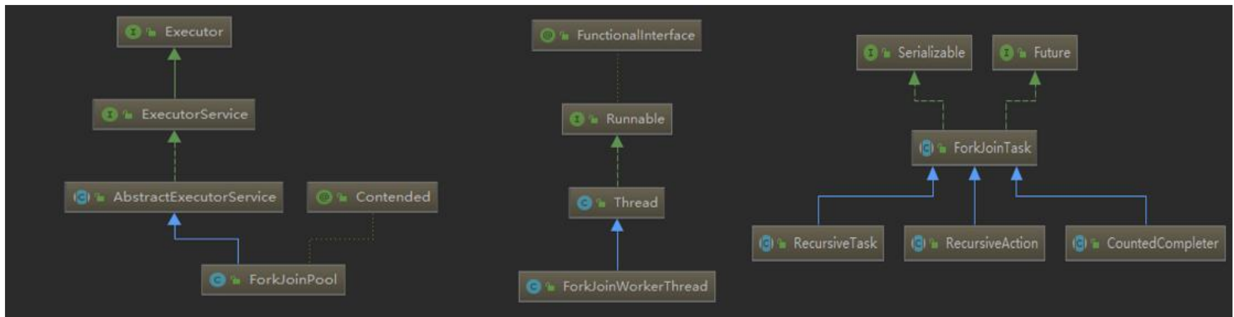
Fork/Join框架局限性：

对于Fork/Join框架而言，当一个任务正在等待它使用Join操作创建的子任务结束时，执行这个任务的工作线程查找其他未被执行的任务，并开始执行这些未被执行的任务，通过这种方式，线程充分利用它们的运行时间来提高应用程序的性能。为了实现这个目标，Fork/Join框架执行的任务有一些局限性。

（1）任务只能使用Fork和Join操作来进行同步机制，如果使用了其他同步机制，则在同步操作时，工作线程就不能执行其他任务了。比如，在Fork/Join框架中，使任务进行了睡眠，那么，在睡眠期间内，正在执行这个任务的工作线程将不会执行其他任务了。
（2）在Fork/Join框架中，所拆分的任务不应该去执行IO操作，比如：读写数据文件。
（3）任务不能抛出检查异常，必须通过必要的代码来出来这些异常。

ForkJoin框架的实现

ForkJoin框架中一些重要的类如下所示。



ForkJoinPool 框架中涉及的主要类如下所示。

1.ForkJoinPool类

实现了ForkJoin框架中的线程池，由类图可以看出，ForkJoinPool类实现了线程池的Executor接口。

我们也可以从下图中看出ForkJoinPool的类图关系。

<<Java Interface>>
Executor

<<Java Interface>>
ExecutorService

<<Java Class>>
AbstractExecutorService

<<Java Class>>
ThreadPoolExecutor

<<Java Class>>
ForkJoinPool

<<Java Class>>
ScheduledThreadPoolExecutor

其中，可以使用Executors.newWorkStealPool()方法创建ForkJoinPool。

ForkJoinPool中提供了如下提交任务的方法。

```
public void execute(ForkJoinTask<?> task)
public void execute(Runnable task)
public <T> T invoke(ForkJoinTask<T> task)
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)
public <T> ForkJoinTask<T> submit(Callable<T> task)
public <T> ForkJoinTask<T> submit(Runnable task, T result)
public ForkJoinTask<?> submit(Runnable task)
```

2.ForkJoinWorkerThread类

实现ForkJoin框架中的线程。

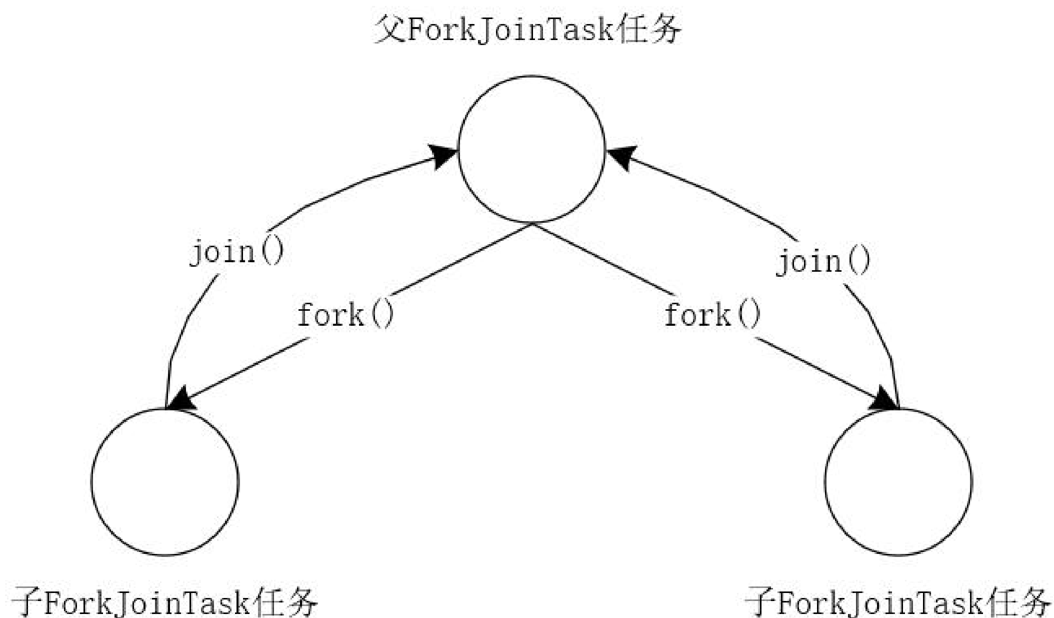
3.ForkJoinTask类

ForkJoinTask封装了数据及其相应的计算，并且支持细粒度的数据并行。ForkJoinTask比线程要轻量，ForkJoinPool中少量工作线程能够运行大量的ForkJoinTask。

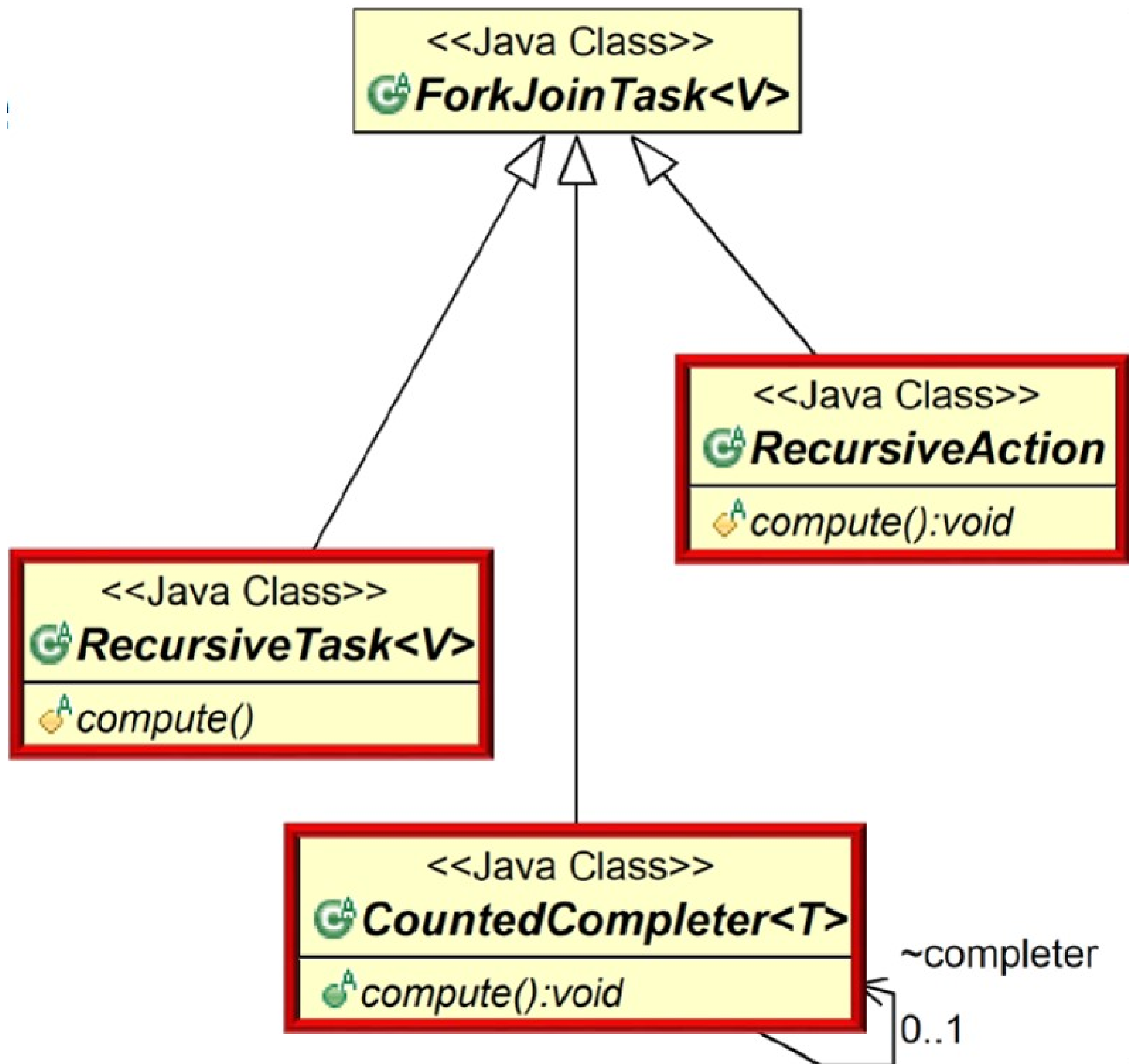
ForkJoinTask类中主要包括两个方法fork()和join()，分别实现任务的分拆与合并。

fork()方法类似于Thread.start()，但是它并不立即执行任务，而是将任务放入工作队列中。跟Thread.join()方法不同，ForkJoinTask的join()方法并不简单的阻塞线程，而是利用工作线程运行其他任务，当一个工作线程中调用join()，它将处理其他任务，直到注意到目标子任务已经完成。

我们可以使用下图来表示这个过程。



ForkJoinTask有3个子类：



- RecursiveAction：无返回值的任务。
- RecursiveTask：有返回值的任务。
- CountedCompleter：完成任务后将触发其他任务。

4.RecursiveTask 类

有返回结果的ForkJoinTask实现Callable。

5.RecursiveAction类

无返回结果的ForkJoinTask实现Runnable。

6.CountedCompleter 类

在任务完成执行后会触发执行一个自定义的钩子函数。

ForkJoin示例程序

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.RecursiveTask;
@Slf4j
public class ForkJoinTaskExample extends RecursiveTask<Integer> {
    public static final int threshold = 2;
    private int start;
    private int end;
    public ForkJoinTaskExample(int start, int end) {
        this.start = start;
```

```java
        this.end = end;
    }
    @Override
    protected Integer compute() {
        int sum = 0;
        //如果任务足够小就计算任务
        boolean canCompute = (end - start) <= threshold;
        if (canCompute) {
            for (int i = start; i <= end; i++) {
                sum += i;
            }
        } else {
            // 如果任务大于阈值，就分裂成两个子任务计算
            int middle = (start + end) / 2;
            ForkJoinTaskExample leftTask = new ForkJoinTaskExample(start, middle);
            ForkJoinTaskExample rightTask = new ForkJoinTaskExample(middle + 1, end);

            // 执行子任务
            leftTask.fork();
            rightTask.fork();

            // 等待任务执行结束合并其结果
            int leftResult = leftTask.join();
            int rightResult = rightTask.join();

            // 合并子任务
            sum = leftResult + rightResult;
        }
        return sum;
    }
    public static void main(String[] args) {
        ForkJoinPool forkjoinPool = new ForkJoinPool();

        //生成一个计算任务，计算1+2+3+4
        ForkJoinTaskExample task = new ForkJoinTaskExample(1, 100);

        //执行一个任务
        Future<Integer> result = forkjoinPool.submit(task);

        try {
            log.info("result:{}", result.get());
        } catch (Exception e) {
            log.error("exception", e);
        }
    }
}
```

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

你知道吗？大家都在使用Redisson实现分布式锁了！！

写在前面

> 忘记之前在哪个群里有朋友在问：有出分布式锁的文章吗~@冰河？我的回答是：这周会有，也是【高并发】专题的。想了想，还是先发一个如何使用Redisson实现分布式锁的文章吧？为啥？因为使用Redisson实现分布式锁简单啊！Redisson框架是基于Redis实现的分布式锁，非常强大，只需要拿来使用就行了，至于分布式锁的原理啥的，后面再撸一篇文章就是了。

Redisson框架十分强大，基于Redisson框架可以实现几乎你能想到的所有类型的分布式锁。这里，我就列举几个类型的分布式锁，并各自给出一个示例程序来加深大家的理解。有关分布式锁的原理细节，后续专门撸一篇文章咱们慢慢聊！

可重入锁（Reentrant Lock）

Redisson的分布式可重入锁RLock Java对象实现了java.util.concurrent.locks.Lock接口，同时还支持自动过期解锁。

```java
public void testReentrantLock(RedissonClient redisson){
    RLock lock = redisson.getLock("anyLock");
```

```
    try{
        // 1. 最常见的使用方法
        //lock.lock();
        // 2. 支持过期解锁功能,10秒钟以后自动解锁, 无需调用unlock方法手动解锁
        //lock.lock(10, TimeUnit.SECONDS);
        // 3. 尝试加锁，最多等待3秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(3, 10, TimeUnit.SECONDS);
        if(res){ //成功
        // do your business
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

Redisson同时还为分布式锁提供了异步执行的相关方法:

```
public void testAsyncReentrantLock(RedissonClient redisson){
    RLock lock = redisson.getLock("anyLock");
    try{
        lock.lockAsync();
        lock.lockAsync(10, TimeUnit.SECONDS);
        Future<Boolean> res = lock.tryLockAsync(3, 10, TimeUnit.SECONDS);
        if(res.get()){
        // do your business
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

公平锁（Fair Lock）

Redisson分布式可重入公平锁也是实现了java.util.concurrent.locks.Lock接口的一种RLock对象。在提供了自动过期解锁功能的同时，保证了当多个Redisson客户端线程同时请求加锁时，优先分配给先发出请求的线程。

```
public void testFairLock(RedissonClient redisson){
    RLock fairLock = redisson.getFairLock("anyLock");
    try{
        // 最常见的使用方法
        fairLock.lock();
        // 支持过期解锁功能, 10秒钟以后自动解锁,无需调用unlock方法手动解锁
        fairLock.lock(10, TimeUnit.SECONDS);
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        fairLock.unlock();
    }
}
```

Redisson同时还为分布式可重入公平锁提供了异步执行的相关方法:

```
RLock fairLock = redisson.getFairLock("anyLock");
fairLock.lockAsync();
fairLock.lockAsync(10, TimeUnit.SECONDS);
Future<Boolean> res = fairLock.tryLockAsync(100, 10, TimeUnit.SECONDS);
```

联锁（MultiLock）

Redisson的RedissonMultiLock对象可以将多个RLock对象关联为一个联锁，每个RLock对象实例可以来自于不同的Redisson实例。

```java
public void testMultiLock(RedissonClient redisson1,RedissonClient redisson2, RedissonClient redisson3){
    RLock lock1 = redisson1.getLock("lock1");
    RLock lock2 = redisson2.getLock("lock2");
    RLock lock3 = redisson3.getLock("lock3");
    RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);
    try {
        // 同时加锁：lock1 lock2 lock3, 所有的锁都上锁成功才算成功。
        lock.lock();
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

红锁（RedLock）

Redisson的RedissonRedLock对象实现了Redlock介绍的加锁算法。该对象也可以用来将多个RLock对象关联为一个红锁，每个RLock对象实例可以来自于不同的Redisson实例。

```java
public void testRedLock(RedissonClient redisson1,RedissonClient redisson2, RedissonClient redisson3){
    RLock lock1 = redisson1.getLock("lock1");
    RLock lock2 = redisson2.getLock("lock2");
    RLock lock3 = redisson3.getLock("lock3");
    RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);
    try {
        // 同时加锁：lock1 lock2 lock3, 红锁在大部分节点上加锁成功就算成功。
        lock.lock();
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

读写锁（ReadWriteLock）

Redisson的分布式可重入读写锁RReadWriteLock,Java对象实现了java.util.concurrent.locks.ReadWriteLock接口。同时还支持自动过期解锁。该对象允许同时有多个读取锁，但是最多只能有一个写入锁。

```java
RReadWriteLock rwlock = redisson.getLock("anyRWLock");
// 最常见的使用方法
rwlock.readLock().lock();
// 或
rwlock.writeLock().lock();
// 支持过期解锁功能
// 10秒钟以后自动解锁
// 无需调用unlock方法手动解锁
rwlock.readLock().lock(10, TimeUnit.SECONDS);
// 或
rwlock.writeLock().lock(10, TimeUnit.SECONDS);
// 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);
// 或
boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);
...
lock.unlock();
```

信号量（Semaphore）

Redisson的分布式信号量（Semaphore）Java对象RSemaphore采用了与java.util.concurrent.Semaphore相似的接口和用法。

```java
RSemaphore semaphore = redisson.getSemaphore("semaphore");
semaphore.acquire();
```

```
//或
semaphore.acquireAsync();
semaphore.acquire(23);
semaphore.tryAcquire();
//或
semaphore.tryAcquireAsync();
semaphore.tryAcquire(23, TimeUnit.SECONDS);
//或
semaphore.tryAcquireAsync(23, TimeUnit.SECONDS);
semaphore.release(10);
semaphore.release();
//或
semaphore.releaseAsync();
```

可过期性信号量（PermitExpirableSemaphore）

Redisson的可过期性信号量（PermitExpirableSemaphore）实在RSemaphore对象的基础上，为每个信号增加了一个过期时间。每个信号可以通过独立的ID来辨识，释放时只能通过提交这个ID才能释放。

```
RPermitExpirableSemaphore semaphore = redisson.getPermitExpirableSemaphore("mySemaphore");
String permitId = semaphore.acquire();
// 获取一个信号，有效期只有2秒钟。
String permitId = semaphore.acquire(2, TimeUnit.SECONDS);
// ...
semaphore.release(permitId);
```

闭锁（CountDownLatch）

Redisson的分布式闭锁（CountDownLatch）Java对象RCountDownLatch采用了与java.util.concurrent.CountDownLatch相似的接口和用法。

```
RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");
latch.trySetCount(1);
latch.await();
// 在其他线程或其他JVM里
RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");
latch.countDown();
```

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

最后，附上并发编程需要掌握的核心技能知识图，祝大家在学习并发编程时，少走弯路。

为何高并发系统中都要使用消息队列?

写在前面

> 很多高并发系统中都会使用到消息队列中间件，那么，问题来了，为什么在高并发系统中都会使用到消息队列中间件呢？立志成为资深架构师的你思考过这个问题吗？
>
> 本文集结了众多技术大牛的编程思想，由冰河汇聚并整理而成，在此，感谢那些在技术发展道理上默默付出的前辈们！

场景分析

现在假设这样一个场景，用户下单成功需要给用户发短信，如果没有消息队列，我们会选择同步调用发短信的接口并等待短信发送成功。现在假设短信接口实现出现了问题或者短信发送短时间内达到了上限，这个时候是选择重试几次还是放弃发送呢？这里的设计会很复杂。如果使用了消息队列，我们选择将发短信的操作封装成一条消息发送到消息队列，消息队列通知一个服务去发送一条短信，即使出现了上述的问题，可以选择把消息重新放到消息队列里等待处理。

消息队列的好处

通过上述了例子，我们看到消息队列完成了一个异步解耦的过程，短信发送时我们只要保证短信发到消息队列成功就可以了，接下来就可以去做别的事情；其次，设计变得更简单，在下单的场景下，我们不用过多考虑发送短信的问题，交给消息队列管理就行了，可能短信发送会有延迟，但是保证了**最终的一致性。**

消息队列特性

- 业务无关，只做消息分发。
- FIFO，先投递先到达。
- 容灾：节点动态增删和消息持久化。
- 性能：吞吐量提升，系统内部通信效率提高

高并发系统为何使用消息队列？

（1）业务解耦

成功完成了一个异步解耦的过程。短信发送时只要保证放到消息队列中就可以了，接着做后面的事情就行。一个事务只关心本质的流程，需要依赖其他事情但是不那么重要的时候，有通知即可，无需等待结果。每个成员不必受其他成员影响，可以更独立自主，只通过一个简单的容器来联系。

对于我们的订单系统，订单最终支付成功之后可能需要给用户发送短信积分什么的，但其实这已经不是我们系统的核心流程了。如果外部系统速度偏慢（比如短信网关速度不好），那么主流程的时间会加长很多，用户肯定不希望点击支付过好几分钟才看到结果。那么我们只需要通知短信系统“我们支付成功了”，不一定非要等待它处理完成。

（2）最终一致性

主要是用记录和补偿的方式来处理；在做所有的不确定事情之前，先把事情记录下来，然后去做不确定的事，它的结果通常分为三种：成功，失败或者不确定；如果成功，我们就可以把记录的东西清理掉，对于失败和不确定，我们可以采用定时任务的方式把所有失败的事情重新做一遍直到成功为止。

保证了最终一致性，通过在队列中存放任务保证它最终一定会执行。

最终一致性指的是两个系统的状态保持一致，要么都成功，要么都失败。当然有个时间限制，理论上越快越好，但实际上在各种异常的情况下，可能会有一定延迟达到最终一致状态，但最后两个系统的状态是一样的。
业界有一些为“最终一致性”而生的消息队列，如Notify（阿里）、QMQ（去哪儿）等，其设计初衷，就是为了交易系统中的高可靠通知。

以一个银行的转账过程来理解最终一致性，转账的需求很简单，如果A系统扣钱成功，则B系统加钱一定成功。反之则一起回滚，像什么都没发生一样。

然而，这个过程中存在很多可能的意外：

- A扣钱成功，调用B加钱接口失败。
- A扣钱成功，调用B加钱接口虽然成功，但获取最终结果时网络异常引起超时。
- A扣钱成功，B加钱失败，A想回滚扣的钱，但A机器down机。

可见，想把这件看似简单的事真正做成，真的不那么容易。所有跨JVM的一致性问题，从技术的角度讲通用的解决方案是：

- 强一致性，分布式事务，但落地太难且成本太高。
- 最终一致性，主要是用“记录”和“补偿”的方式。在做所有的不确定的事情之前，先把事情记录下来，然后去做不确定的事情，结果可能是：成功、失败或是不确定，“不确定”（例如超时等）可以等价为失败。成功就可以把记录的东西清理掉了，对于失败和不确定，可以依靠定时任务等方式把所有失败的事情重新搞一遍，直到成功为止。

回到刚才的例子，系统在A扣钱成功的情况下，把要给B“通知”这件事记录在库里（为了保证最高的可靠性可以把通知B系统加钱和扣钱成功这两件事维护在一个本地事务里），通知成功则删除这条记录，通知失败或不确定则依靠定时任务补偿性地通知我们，直到我们把状态更新成正确的为止。

消息可能重复，注意消息的重复和幂等。

（3）广播

如果没有消息队列，每当一个新的业务接入时，我们都需要连接一个新接口；有了消息队列，我们只需要关系消息是否送到到消息队列，新接入的接口订阅相关的消息，自己去做处理就行了。

（4）错峰与流控

利用消息队列，转储两个系统的通信内容，并在下游系统有能力处理这些消息的时候再处理这些消息。试想上下游对于事情的处理能力是不同的。比如，Web前端每秒承受上千万的请求，并不是什么神奇的事情，只需要加多一点机器，再搭建一些LVS负载均衡设备和Nginx等即可。但数据库的处理能力却十分有限，即使使用SSD加分库分表，单机的处理能力仍然在万级。由于成本的考虑，我们不能奢求数据库的机器数量追上前端。

这种问题同样存在于系统和系统之间，如短信系统可能由于短板效应，速度卡在网关上（每秒几百次请求），跟前端的并发量不是一个数量级。但用户晚上个半分钟左右收到短信，一般是不会有太大问题的。如果没有消息队列，两个系统之间通过协商、滑动窗口等复杂的方案也不是说不能实现。但系统复杂性指数级增长，势必在上游或者下游做存储，并且要处理定时、拥塞等一系列问题。而且每当有处理能力有差距的时候，都需要单独开发一套逻辑来维护这套逻辑。所以，利用中间系统转储两个系统的通信内容，并在下游系统有能力处理这些消息的时候，再处理这些消息，是一套相对较通用的方式。

总结

总而言之，消息队列不是万能的。对于需要强事务保证而且延迟敏感的，RPC是优于消息队列的。

对于一些无关痛痒，或者对于别人非常重要但是对于自己不是那么关心的事情，可以利用消息队列去做。

支持最终一致性的消息队列，能够用来处理延迟不那么敏感的"分布式事务"场景，而且相对于笨重的分布式事务，可能是更优的处理方式。

当上下游系统处理能力存在差距的时候，利用消息队列做一个通用的"漏斗"。在下游有能力处理的时候，再进行分发。
如果下游有很多系统关心你的系统发出的通知的时候，果断地使用消息队列吧。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

高并发环境下如何优化Tomcat配置？看完我懂了！

写在前面

> Tomcat作为最常用的Java Web服务器，随着并发量越来越高，Tomcat的性能会急剧下降，那有没有什么方法来优化Tomcat在高并发环境下的性能呢?

Tomcat运行模式

Tomcat的运行模式有3种。

1.bio模式

默认的模式,性能非常低下,没有经过任何优化处理和支持。

2.nio模式

利用java的异步io护理技术,noblocking IO技术。要想运行在该模式下，则直接修改server.xml里的Connector节点,修改protocol为如下配置。

```
protocol="org.apache.coyote.http11.Http11NioProtocol"
```

重启Tomcat后，就可以生效。

3.apr模式

安装起来最困难,但是从操作系统级别来解决异步的IO问题,大幅度的提高性能。此种模式下，必须要安装apr和native，直接启动就支持apr。如nio修改模式，修改protocol为org.apache.coyote.http11.Http11AprProtocol，如下所示。

```
protocol="org.apache.coyote.http11.Http11AprProtocol"
```

Tomcat并发优化

安装APR

```
[root@binghe ~]# yum -y install apr apr-devel openssl-devel
[root@binghe ~]# tar zxvf tomcat-native.tar.gz
[root@binghe ~]# cd tomcat-native-1.1.24-src/jni/native
[root@binghe native]# ./configure --with-apr=/usr/bin/apr-1-config --with-ssl=/usr/include/openssl/
[root@binghe native]# make && make install
```

安装完成之后 会出现如下提示信息

```
Libraries have been installed in:
/usr/local/apr/lib
```

安装成功后还需要对tomcat设置环境变量，方法是在catalina.sh文件中增加1行：

在这段代码下面添加：

```
============
# OS specific support.  $var _must_ be set to either true or false.
cygwin=false
darwin=false
==============
CATALINA_OPTS="-Djava.library.path=/usr/local/apr/lib"
```

修改server.xml的配置，如下所示。

```
protocol="org.apache.coyote.http11.Http11AprProtocol"
```

启动tomcat之后，查看日志，如下所示。

```
more TOMCAT_HOME/logs/catalina.out
2020-04-17 22:34:56 org.apache.catalina.core.AprLifecycleListener init
INFO: Loaded APR based Apache Tomcat Native library 1.1.31 using APR version 1.3.9.
2020-04-17 22:34:56 org.apache.catalina.core.AprLifecycleListener init
INFO: APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true].
2020-04-17 22:34:56 org.apache.catalina.core.AprLifecycleListener initializeSSL
INFO: OpenSSL successfully initialized (OpenSSL 1.0.1e 11 Feb 2013)
2020-04-17 22:34:58 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-apr-8080"]
2020-04-17 22:34:58 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["ajp-apr-8009"]
2020-04-17 22:34:58 AM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1125 ms
```

Tomcat优化

1.JVM 调优

在TOMCAT_HOME/bin/catalina.sh 增加如下语句，具体数值视情况而定。
添加到上面CATALINA_OPTS的后面即可，如下所示。

```
JAVA_OPTS=-Xms512m -Xmx1024m -XX:PermSize=512M -XX:MaxNewSize=1024m -XX:MaxPermSize=1024m
```

参数详解

- -Xms：JVM初始化堆内存大小。
- -Xmx：JVM堆的最大内存。
- -Xss：线程栈大小。
- -XX:PermSize：JVM非堆区初始内存分配大小。
- -XX:MaxPermSize：JVM非堆区最大内存。

建议和注意事项:

-Xms和-Xmx选项设置为相同堆内存分配，以避免在每次GC 后调整堆的大小，堆内存建议占内存的60%~80%；非堆内存是不可回收内存，大小视项目而定;线程栈大小推荐256k。

32G内存配置如下：

```
JAVA_OPTS=-Xms20480m -Xmx20480m -Xss1024K -XX:PermSize=512m -XX:MaxPermSize=2048m
```

2.关闭DNS反向查询

在<Connector port="8080" 中加入如下参数。

```
enableLookups="false"
```

3.优化tomcat参数

在server.xml文件中进行如下配置。

```
<Connector port="8080"
protocol="org.apache.coyote.http11.Http11AprProtocol"
connectionTimeout="20000" //链接超时时长
redirectPort="8443"
maxThreads="500"//设定处理客户请求的线程的最大数目，决定了服务器可以同时响应客户请求的数,默认200
minSpareThreads="20"//初始化线程数，最小空闲线程数，默认为10
acceptCount="1000" //当所有可以使用的处理请求的线程数都被使用时，可以被放到处理队列中请求数，请求数超过这个数的请求将不予处理，默认100
enableLookups="false"
URIEncoding="UTF-8" />
```

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

不废话，言简意赅介绍BlockingQueue

写在前面

> 最近，有不少网友留言提问：在Java的并发专题中，有个BlockingQueue，它是个阻塞队列，为何要在并发编程里使用BlockingQueue呢？好吧，今天，就临时说一下BlockingQueue吧，不过今天说的不是很深入，后面咱们一起从源头上深入剖析这个类。

BlockingQueue概述

阻塞队列，是线程安全的。

被阻塞的情况如下：

（1）当队列满时，进行入队列操作
（2）当队列空时，进行出队列操作

使用场景如下：

主要在生产者和消费者场景。

BlockingQueue的方法

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除	remove()	poll()	take()	poll(time, unit)
检查	element()	peek()	不可用	不可用

四组不同的行为方式解释

- 抛出异常

如果试图的操作无法立即执行，抛一个异常。

- 特殊值

如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

- 阻塞

如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

- 超时

如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

BlockingQueue的实现类

- ArrayBlockingQueue：有界的阻塞队列（容量有限，必须在初始化的时候指定容量大小，容量大小指定后就不能再变化），内部实现是一个数组，以FIFO的方式存储数据，最新插入的对象是尾部，最新移除的对象是头部。
- DelayQueue：阻塞的是内部元素，DelayQueue中的元素必须实现一个接口——Delayed（存在于J.U.C下）。Delayed接口继承了Comparable接口，这是因为Delayed接口中的元素需要进行排序，一般情况下，都是按照Delayed接口中的元素过期时间的优先级进行排序。应用场景主要有：定时关闭连接、缓存对象、超时处理等。内部实现使用PriorityQueue和ReentrantLock。
- LinkedBlockingQueue：大小配置是可选的，如果初始化时指定了大小，则是有边界的；如果初始化时未指定大小，则是无边界的（其实默认大小是Integer类型的最大值）。内部实现时一个链表，以FIFO的方式存储数据，最新插入的对象是尾部，最新移除的对象是头部。
- PriorityBlockingQueue：带优先级的阻塞队列，无边界，但是有排序规则，允许插入空对象(也就是null)。所有插入的对象必须实现Comparable接口，队列优先级的排序规则就是按照对Comparable接口的实现来定义的。可以从PriorityBlockingQueue中获得一个迭代器Iterator，但这个迭代器并不保证按照优先级的顺序进行迭代。
- SynchronousQueue：队列内部仅允许容纳一个元素，当一个线程插入一个元素后，就会被阻塞，除非这个元素被另一个线程消费。因此，也称SynchronousQueue为同步队列。SynchronousQueue是一个无界非缓存的队列。准确的说，它不存储元素，放入元素只有等待取走元素之后，才能再次放入元素，

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

高并发环境下如何防止Tomcat内存溢出？

写在前面

> 随着系统并发量越来越高，Tomcat所占用的内存就会越来越大，如果对Tomcat的内存管理不当，则可能会引发Tomcat内存溢出的问题，那么，如何防止Tomcat内存溢出呢？我们今天就来一起探讨下这个问题。

防止Tomcat内存溢出可以总结为两个方案：一个是设置Tomcat启动的初始内存，一个是防止Tomcat所用的JVM内存溢出。接下来，我们就分别对这两种方案作出简单的介绍。

设置启动初始内存

其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。

实例

以下给出1G内存环境下java jvm 的参数设置参考：

```
JAVA_OPTS="-server -Xms800m -Xmx800m  -XX:PermSize=64M -XX:MaxNewSize=256m -XX:MaxPermSize=128m -Djava.awt.headless=true "
JAVA_OPTS="-server -Xms768m -Xmx768m -XX:PermSize=128m -XX:MaxPermSize=256m -XX:NewSize=192m -XX:MaxNewSize=384m"
CATALINA_OPTS="-server -Xms768m -Xmx768m -XX:PermSize=128m -XX:MaxPermSize=256m -XX:NewSize=192m -XX:MaxNewSize=384m"
```

Linux

在/usr/local/apache-tomcat-7.0/bin 目录下的catalina.sh文件中，添加：JAVA_OPTS='-Xms512m -Xmx1024m'，要加“m”说明是MB，否则就是KB了，在启动tomcat时会报内存不足。

- -Xms：初始值
- -Xmx：最大值
- -Xmn：最小值

Windows

在catalina.bat最前面加入set JAVA_OPTS=-Xms128m -Xmx350m，如果用startup.bat启动tomcat,OK设置生效。够成功的分配200M内存。但是如果不是执行startup.bat启动tomcat而是利用windows的系统服务启动tomcat服务,上面的设置就不生效了，就是说set JAVA_OPTS=-Xms128m -Xmx350m 没起作用。上面分配200M内存就OOM了。。
windows服务执行的是bin\tomcat.exe。它读取注册表中的值，而不是catalina.bat的设置。

解决办法

修改注册表

```
HKEY_LOCAL_MACHINE\SOFTWARE\Apache Software Foundation\Tomcat Service
Manager\Tomcat5\Parameters\JavaOptions
```

原值为

```
-Dcatalina.home="C:\ApacheGroup\Tomcat 7.0"
-Djava.endorsed.dirs="C:\ApacheGroup\Tomcat 7.0\common\endorsed"
-Xrs
```

加入 -Xms300m -Xmx350m
重起tomcat服务，设置生效。

防止所用的JVM内存溢出

1.java.lang.OutOfMemoryError: Java heap space

解释

Heap size 设置

JVM堆的设置是指java程序运行过程中JVM可以调配使用的内存空间的设置.JVM在启动的时候会自动设置Heap size的值，其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap size 的大小是Young Generation 和Tenured Generaion 之和。

提示：在JVM中如果98％的时间是用于GC且可用的Heap size 不足2％的时候将抛出此异常信息。

提示：Heap Size 最大不要超过可用物理内存的80％，一般的要将-Xms和-Xmx选项设置为相同，而-Xmn为1/4的-Xmx值。

解决方法

手动设置Heap size

修改TOMCAT_HOME/bin/catalina.bat，在“echo "Using CATALINA_BASE: $CATALINA_BASE"”上面加入以下代码。

```
set JAVA_OPTS=%JAVA_OPTS% -server -Xms800m -Xmx800m -XX:MaxNewSize=256m
set JAVA_OPTS=%JAVA_OPTS% -server -Xms800m -Xmx800m -XX:MaxNewSize=256m
```

或修改catalina.sh，在“echo "Using CATALINA_BASE: $CATALINA_BASE"”上面加入以下行：

```
JAVA_OPTS="$JAVA_OPTS -server -Xms800m -Xmx800m -XX:MaxNewSize=256m"
```

2.java.lang.OutOfMemoryError: PermGen space

原因

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域，这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGen space中，它和存放类实例(Instance)的Heap区域不同,GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的应用中有很CLASS的话,就很可能出现PermGen space错误，这种错误常见在web服务器对JSP进行pre compile的时候。如果你的WEB APP下都用了大量的第三方jar, 其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。

解决方法

手动设置MaxPermSize大小

修改TOMCAT_HOME/bin/catalina.bat（Linux下为catalina.sh），在代码“echo "Using CATALINA_BASE: $CATALINA_BASE"”上面加入以下行：

```
set JAVA_OPTS=%JAVA_OPTS% -server -XX:PermSize=128M -XX:MaxPermSize=512m
```

“echo "Using CATALINA_BASE: $CATALINA_BASE"”上面加入以下行：

```
set JAVA_OPTS=%JAVA_OPTS% -server -XX:PermSize=128M -XX:MaxPermSize=512m
```

catalina.sh文件的修改如下。

Java代码

```
JAVA_OPTS="$JAVA_OPTS -server -XX:PermSize=128M -XX:MaxPermSize=512m"
```

3.分析java.lang.OutOfMemoryError: PermGen space

发现很多人把问题归因于： spring,hibernate,tomcat，因为他们动态产生类,导致JVM中的permanent heap溢出 。然后解决方法众说纷纭，有人说升级 tomcat版本到最新甚至干脆不用tomcat。还有人怀疑spring的问题，在spring论坛上讨论很激烈，因为spring在AOP时使用CBLIB会动态产生很多类。

但问题是为什么这些王牌的开源会出现同一个问题呢，那么是不是更基础的原因呢？tomcat在Q&A很隐晦的回答了这一点，我们知道这个问题，但这个问题是由一个更基础的问题产生。

于是有人对更基础的JVM做了检查，发现了问题的关键。原来SUN 的JVM把内存分了不同的区，其中一个就是permenter区用来存放用得非常多的类和类描述。本来SUN设计的时候认为这个区域在JVM启动的时候就固定了，但他没有想到现在动态会用得这么广泛。而且这个区域有特殊的垃圾收回机制，现在的问题是动态加载类到这个区域后，gc根本没办法回收！

对于以上两个问题，我的处理是：

在catalina.bat的第一行增加：

```
set JAVA_OPTS=-Xms64m -Xmx256m -XX:PermSize=128M -XX:MaxNewSize=256m -XX:MaxPermSize=256m
```

在catalina.sh的第一行增加：

```
JAVA_OPTS= -Xms64m -Xmx256m -XX:PermSize=128M -XX:MaxNewSize=256m -XX:MaxPermSize=256m
```

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

高并发下常见的限流方案

写在前面

> 本文转自涛哥的《亿级流量网站架构核心技术》，啥也不说了，向开涛大神致敬！

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。缓存的目的是提升系统访问速度和增大系统能处理的容量，可谓是抗高并发流量的银弹；而降级是当服务出问题或者影响到核心流程的性能则需要暂时屏蔽掉，待高峰或者问题解决后再打开；而有些场景并不能用缓存和降级来解决，比如稀缺资源（秒杀、抢购）、写服务（如评论、下单）、频繁的复杂查询（评论的最后几页），因此需有一种手段来限制这些场景的并发/请求量，即限流。

限流的目的是通过对并发访问/请求进行限速或者一个时间窗口内的的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务（定向到错误页或告知资源没有了）、排队或等待（比如秒杀、评论、下单）、降级（返回兜底数据或默认数据，如商品详情页库存默认有货）。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如nginx的limit_conn模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit_req模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制MQ的消费速率。另外还可以根据网络连接数、网络流量、CPU或内存负载等来限流。

先有缓存这个银弹，后有限流来应对618、双十一高并发流量，在处理高并发问题上可以说是如虎添翼，不用担心瞬间流量导致系统挂掉或雪崩，最终做到有损服务而不是不服务；限流需要评估好，不可乱用，否则会正常流量出现一些奇怪的问题而导致用户抱怨。

在实际应用时也不要太纠结算法问题，因为一些限流算法实现是一样的只是描述不一样；具体使用哪种限流技术还是要根据实际场景来选择，不要一味去找最佳模式，白猫黑猫能解决问题的就是好猫。
因在实际工作中遇到过许多人来问如何进行限流，因此本文会详细介绍各种限流手段。那么接下来我们从限流算法、应用级限流、分布式限流、接入层限流来详细学习下限流技术手段。

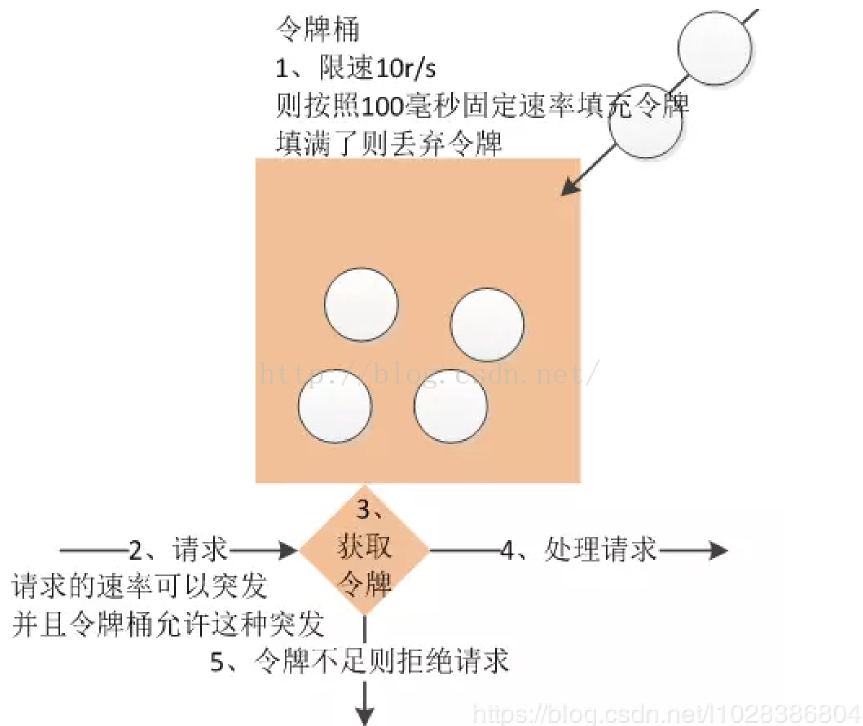
限流算法

常见的限流算法有：令牌桶、漏桶。计数器也可以进行粗暴限流实现

令牌桶算法

令牌桶算法是一个存放固定容量令牌的桶，按照固定速率往桶里添加令牌。令牌桶算法的描述如下：

- 假设限制2r/s，则按照500毫秒的固定速率往桶中添加令牌；
- 桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃或拒绝；
- 当一个n个字节大小的数据包到达，将从桶中删除n个令牌，接着数据包被发送到网络上；
- 如果桶中的令牌不足n个，则不会删除令牌，且该数据包将被限流（要么丢弃，要么缓冲区等待）。



漏桶算法

漏桶作为计量工具（The Leaky Bucket Algorithm as a Meter）时，可以用于流量整形（Traffic Shaping）和流量控制（TrafficPolicing），漏桶算法的描述如下：

- 一个固定容量的漏桶，按照常量固定速率流出水滴；
- 如果桶是空的，则不需流出水滴；
- 可以以任意速率流入水滴到漏桶；
- 如果流入水滴超出了桶的容量，则流入的水滴溢出了（被丢弃），而漏桶容量是不变的。

令牌桶和漏桶对比

- 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，当令牌数减为零时则拒绝新的请求；
- 漏桶则是按照常量固定速率流出请求，流入请求速率任意，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；
- 令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量；
- 漏桶限制的是常量流出速率（即流出速率是一个固定常量值，比如都是1的速率流出，而不能一次是1，下次又是2），从而平滑突发流入速率；
- 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流入速率；
- 两个算法实现可以一样，但是方向是相反的，对于相同的参数得到的限流效果是一样的。

另外有时候我们还使用计数器来进行限流，主要用来限制总并发数，比如数据库连接池、线程池、秒杀的并发数；只要全局总请求数或者一定时间段的总请求数设定的阀值则进行限流，是简单粗暴的总数量限流，而不是平均速率限流。

到此基本的算法就介绍完了，接下来我们首先看看应用级限流。

单机限流

```
// 令牌桶算法实现 tryAcquire
@Slf4j
public class RateLimiterExample1 {

    private static RateLimiter rateLimiter = RateLimiter.create(5);

    public static void main(String[] args) throws Exception {

        for (int index = 0; index < 100; index++) {
            if (rateLimiter.tryAcquire(190, TimeUnit.MILLISECONDS)) {
                handle(index);
            }
        }
    }
```

```java
    private static void handle(int i) {
        log.info("{}", i);
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_162.jdk/Contents/Home/bin/java ...
14:35:23.812 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample1 - 0
14:35:24.012 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample1 - 1

Process finished with exit code 0
```

```java
// 令牌桶算法实现 acquire
@Slf4j
public class RateLimiterExample2 {

    private static RateLimiter rateLimiter = RateLimiter.create(5);

    public static void main(String[] args) throws Exception {

        for (int index = 0; index < 100; index++) {
            rateLimiter.acquire();
            handle(index);
        }
    }

    private static void handle(int i) {
        log.info("{}", i);
    }
}
```

```
14:41:38.052 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 87
14:41:38.253 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 88
14:41:38.452 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 89
14:41:38.651 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 90
14:41:38.851 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 91
14:41:39.054 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 92
14:41:39.255 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 93
14:41:39.454 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 94
14:41:39.651 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 95
14:41:39.852 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 96
14:41:40.054 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 97
14:41:40.253 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 98
14:41:40.451 [main] INFO com.mmall.concurrency.example.rateLimiter.RateLimiterExample2 - 99

Process finished with exit code 0
```

https://blog.csdn.net/l1028386804

应用级限流

限流总并发/连接/请求数

对于一个应用系统来说一定会有极限并发/请求数，即总有一个TPS/QPS阀值，如果超了阀值则系统就会不响应用户请求或响应的非常慢，因此我们最好进行过载保护，防止大量请求涌入击垮系统。

如果你使用过Tomcat，其Connector其中一种配置有如下几个参数：

- acceptCount：如果Tomcat的线程都忙于响应，新来的连接会进入队列排队，如果超出排队大小，则拒绝连接；
- maxConnections：瞬时最大连接数，超出的会排队等待；
- maxThreads：Tomcat能启动用来处理请求的最大线程数，如果请求处理量一直远远大于最大线程数则可能会僵死。

详细的配置请参考官方文档。另外如MySQL（如max_connections）、Redis（如tcp-backlog）都会有类似的限制连接数的配置。

限流总资源数

如果有的资源是稀缺资源（如数据库连接、线程），而且可能有多个系统都会去使用它，那么需要限制应用；可以使用池化技术来限制总资源数：连接池、线程池。比如分配给每个应用的数据库连接是100，那么本应用最多可以使用100个资源，超出了可以等待或者抛异常。

限流某个接口的总并发/请求数

如果接口可能会有突发访问情况，但又担心访问量太大造成崩溃，如抢购业务；这个时候就需要限制这个接口的总并发/请求数总请求数了；因为粒度比较细，可以为每个接口都设置相应的阀值。可以使用Java中的AtomicLong进行限流：

```
try {
    if(atomic.incrementAndGet() > 限流数) {
        //拒绝请求
    }
    //处理请求
} finally {
    atomic.decrementAndGet();
}
```

适合对业务无损的服务或者需要过载保护的服务进行限流，如抢购业务，超出了大小要么让用户排队，要么告诉用户没货了，对用户来说是可以接受的。而一些开放平台也会限制用户调用某个接口的试用请求量，也可以用这种计数器方式实现。这种方式也是简单粗暴的限流，没有平滑处理，需要根据实际情况选择使用；

限流某个接口的时间窗请求数

即一个时间窗口内的请求数，如想限制某个接口/服务每秒/每分钟/每天的请求数/调用量。如一些基础服务会被很多其他系统调用，比如商品详情页服务会调用基础商品服务调用，但是怕因为更新量比较大将基础服务打挂，这时我们要对每秒/每分钟的调用量进行限速；一种实现方式如下所示

```
LoadingCache<Long, AtomicLong> counter =
        CacheBuilder.newBuilder()
                .expireAfterWrite(2, TimeUnit.SECONDS)
                .build(new CacheLoader<Long, AtomicLong>() {
                    @Override
                    public AtomicLong load(Long seconds) throws Exception {
                        return new AtomicLong(0);
                    }
                });
long limit = 1000;
while(true) {
    //得到当前秒
    long currentSeconds = System.currentTimeMillis() / 1000;
    if(counter.get(currentSeconds).incrementAndGet() > limit) {
        System.out.println("限流了:" + currentSeconds);
        continue;
    }
    //业务处理
}
```

我们使用Guava的Cache来存储计数器，过期时间设置为2秒（保证1秒内的计数器是有的），然后我们获取当前时间戳然后取秒数来作为KEY进行计数统计和限流，这种方式也是简单粗暴，刚才说的场景够用了。

平滑限流某个接口的请求数

之前的限流方式都不能很好地应对突发请求，即瞬间请求可能都被允许从而导致一些问题；因此在一些场景中需要对突发请求进行整形，整形为平均速率请求处理（比如5r/s，则每隔200毫秒处理一个请求，平滑了速率）。这个时候有两种算法满足我们的场景：令牌桶和漏桶算法。Guava框架提供了令牌桶算法实现，可直接拿来使用。

Guava RateLimiter提供了令牌桶算法实现：平滑突发限流(SmoothBursty)和平滑预热限流(SmoothWarmingUp)实现。

SmoothBursty

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

将得到类似如下的输出：

```
0.0
0.198239
0.196083
0.200609
0.199599
0.19961
```

- RateLimiter.create(5)表示桶容量为5且每秒新增5个令牌，即每隔200毫秒新增一个令牌；
- limiter.acquire()表示消费一个令牌，如果当前桶中有足够令牌则成功（返回值为0），如果桶中没有令牌则暂停一段时间，比如发令牌间隔是200毫秒，则等待200毫秒后再去消费令牌（如上测试用例返回的为0.198239，差不多等待了200毫秒桶中才有令牌可用），这种实现将突发请求速率平均为了固定请求速率。

再看一个突发示例

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(5));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1))
```

将得到类似如下的输出：

```
0.0
0.98745
0.183553
0.199909
```

limiter.acquire(5)表示桶的容量为5且每秒新增5个令牌，令牌桶算法允许一定程度的突发，所以可以一次性消费5个令牌，但接下来的limiter.acquire(1)将等待差不多1秒桶中才能有令牌，且接下来的请求也整形为固定速率了。

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(10));
System.out.println(limiter.acquire(1));
System.out.println(limiter.acquire(1));
```

将得到类似如下的输出：

```
0.0
1.997428
0.192273
0.200616
```

同上边的例子类似，第一秒突发了10个请求，令牌桶算法也允许了这种突发（允许消费未来的令牌），但接下来的limiter.acquire(1)将等待差不多2秒桶中才能有令牌，且接下来的请求也整形为固定速率了。

接下来再看一个突发的例子：

```
RateLimiter limiter = RateLimiter.create(2);

System.out.println(limiter.acquire());
Thread.sleep(2000L);
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
System.out.println(limiter.acquire());
```

将得到类似如下的输出：

```
0.0
0.0
0.0
0.0
0.499876
0.495799
```

1、创建了一个桶容量为2且每秒新增2个令牌；

2、首先调用limiter.acquire()消费一个令牌，此时令牌桶可以满足（返回值为0）；

3、然后线程暂停2秒，接下来的两个limiter.acquire()都能消费到令牌，第三个limiter.acquire()也同样消费到了令牌，到第四个时就需要等待500毫秒了。

此处可以看到我们设置的桶容量为2（即允许的突发量），这是因为SmoothBursty中有一个参数：最大突发秒数（maxBurstSeconds）默认值是1s，突发量/桶容量=速率*maxBurstSeconds，所以本示例桶容量/突发量为2，例子中前两个是消费了之前积攒的突发量，而第三个开始就是正常计算的了。令牌桶算法允许将一段时间内没有消费的令牌暂存到令牌桶中，留待未来使用，并允许未来请求的这种突发。

SmoothBursty通过平均速率和最后一次新增令牌的时间计算出下次新增令牌的时间的，另外需要一个桶暂存一段时间内没有使用的令牌（即可以突发的令牌数）。另外RateLimiter还提供了tryAcquire方法来进行无阻塞或可超时的令牌消费。

因为SmoothBursty允许一定程度的突发，会有人担心如果允许这种突发，假设突然间来了很大的流量，那么系统很可能扛不住这种突发。因此需要一种平滑速率的限流工具，从而系统冷启动后慢慢的趋于平均固定速率（即刚开始速率小一些，然后慢慢趋于我们设置的固定速率）。Guava也提供了SmoothWarmingUp来实现这种需求，其可以认为是漏桶算法，但是在某些特殊场景又不太一样。

SmoothWarmingUp创建方式：RateLimiter.create(doublepermitsPerSecond, long warmupPeriod, TimeUnit unit) permitsPerSecond表示每秒新增的令牌数，warmupPeriod表示在从冷启动速率过渡到平均速率的时间间隔。

示例如下：

```
RateLimiter limiter = RateLimiter.create(5,1000, TimeUnit.MILLISECONDS);
for(inti =1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
Thread.sleep(1000L);
for(inti =1; i < 5;i++) {
    System.out.println(limiter.acquire());
}
```

将得到类似如下的输出：

```
0.0
0.51767
0.357814
0.219992
0.199984
0.0
0.360826
0.220166
0.199723
0.199555
```

速率是梯形上升速率的，也就是说冷启动时会以一个比较大的速率慢慢到平均速率；然后趋于平均速率（梯形下降到平均速率）。可以通过调节warmupPeriod参数实现一开始就是平滑固定速率。

到此应用级限流的一些方法就介绍完了。假设将应用部署到多台机器，应用级限流方式只是单应用内的请求限流，不能进行全局限流。因此我们需要分布式限流和接入层限流来解决这个问题。

分布式限流

分布式限流最关键的是要将限流服务做成原子化，而解决方案可以使使用Redis+lua或者nginx+lua技术进行实现，通过这两种技术可以实现的高并发和高性能。

首先我们来使用redis+lua实现时间窗内某个接口的请求数限流，实现了该功能后可以改造为限流总并发/请求数和限制总资源数。Lua本身就是一种编程语言，也可以使用它实现复杂的令牌桶或漏桶算法。

redis+lua实现的lua脚本

```
local key = KEYS[1] --限流KEY（一秒一个）
local limit = tonumber(ARGV[1])        --限流大小
local current = tonumber(redis.call("INCRBY", key, "1")) --请求数+1
if current > limit then --如果超出限流大小
   return 0
elseif current == 1 then  --只有第一次访问需要设置2秒的过期时间
   redis.call("expire", key,"2")
end
return 1
```

如上操作因是在一个lua脚本中，又因Redis是单线程模型，因此是线程安全的。如上方式有一个缺点就是当达到限流大小后还是会递增的，可以改造成如下方式实现：

```
local key = KEYS[1] --限流KEY（一秒一个）
local limit = tonumber(ARGV[1])        --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
   return 0
else  --请求数+1，并设置2秒过期
   redis.call("INCRBY", key,"1")
   redis.call("expire", key,"2")
   return 1
end
```

如下是Java中判断是否需要限流的代码：

```
public static booleanacquire()throwsException {
    String luaScript = Files.toString(newFile("limit.lua"), Charset.defaultCharset());
    Jedis jedis = newJedis("192.168.147.52",6379);
    String key = "ip:"+ System.currentTimeMillis()/1000;//此处将当前时间戳取秒数
    Stringlimit ="3";//限流大小
    return(Long)jedis.eval(luaScript,Lists.newArrayList(key), Lists.newArrayList(limit)) ==1;
}
```

因为Redis的限制（Lua中有写操作不能使用带随机性质的读操作，如TIME）不能在Redis Lua中使用TIME获取时间戳，因此只好从应用获取然后传入，在某些极端情况下（机器时钟不准的情况下），限流会存在一些小问题。

使用Nginx+Lua实现的Lua脚本：

```
local locks = require "resty.lock"

local function acquire()
    local lock =locks:new("locks")
    local elapsed, err =lock:lock("limit_key") --互斥锁
    local limit_counter =ngx.shared.limit_counter --计数器

    local key = "ip:" ..os.time()
    local limit = 5 --限流大小
    local current =limit_counter:get(key)

    if current ~= nil and current + 1> limit then --如果超出限流大小
       lock:unlock()
       return 0
    end
    if current == nil then
       limit_counter:set(key, 1, 1) --第一次需要设置过期时间，设置key的值为1，过期时间为1秒
    else
        limit_counter:incr(key, 1) --第二次开始加1即可
    end
    lock:unlock()
    return 1
end
ngx.print(acquire())
```

实现中我们需要使用lua-resty-lock互斥锁模块来解决原子性问题(在实际工程中使用时请考虑获取锁的超时问题)，并使用ngx.shared.DICT共享字典来实现计数器。如果需要限流则返回0，否则返回1。使用时需要先定义两个共享字典（分别用来存放锁和计数器数据）：

```
http {
    ……
    lua_shared_dict locks 10m;
    lua_shared_dict limit_counter 10m;
}
```

有人会纠结如果应用并发量非常大那么redis或者nginx是不是能抗得住；不过这个问题要从多方面考虑：你的流量是不是真的有这么大，是不是可以通过一致性哈希将分布式限流进行分片，是不是可以当并发量太大降级为应用级限流；对策非常多，可以根据实际情况调节；像在京东使用Redis+Lua来限流抢购流量，一般流量是没有问题的。

对于分布式限流目前遇到的场景是业务上的限流，而不是流量入口的限流；流量入口限流应该在接入层完成，而接入层笔者一般使用Nginx。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

Redis如何助力高并发秒杀系统？看完这篇我彻底懂了！！

写在前面

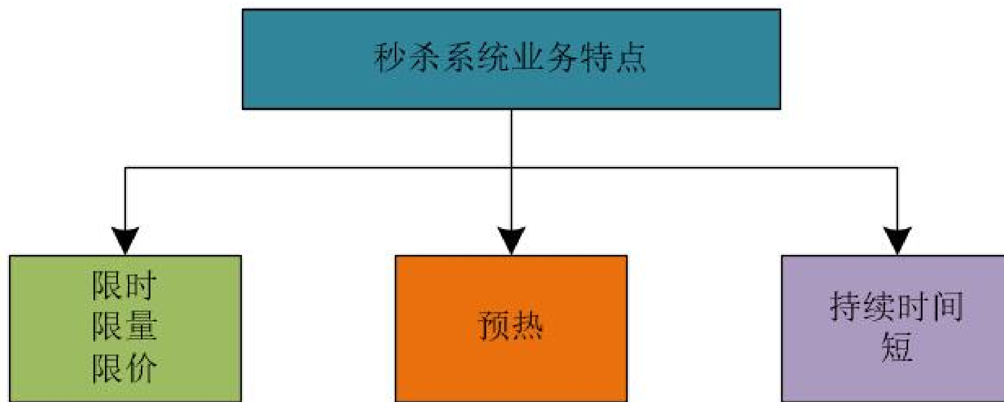
> 之前，我们在《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》一文中，详细讲解了高并发秒杀系统的架构设计，其中，我们介绍了可以使用Redis存储秒杀商品的库存数量。很多小伙伴看完后，觉得一头雾水，看完是看完了，那如何实现呢？今天，我们就一起来看看Redis是如何助力高并发秒杀系统的！
>
> 有关高并发秒杀系统的架构设计，小伙伴们可以关注 **冰河技术** 公众号，查看《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》一文。

秒杀业务

在电商领域，存在着典型的秒杀业务场景，那何谓秒杀场景呢。简单的来说就是一件商品的购买人数远远大于这件商品的库存，而且这件商品在很短的时间内就会被抢购一空。比如每年的618、双11大促，小米新品促销等业务场景，就是典型的秒杀业务场景。

秒杀业务最大的特点就是瞬时并发流量高，在电商系统中，库存数量往往会远远小于并发流量，比如：天猫的秒杀活动，可能库存只有几百、几千件，而瞬间涌入的抢购并发流量可能会达到几十到几百万。

所以，我们可以将秒杀系统的业务特点总结如下。



（1）限时、限量、限价

在规定的时间内进行；秒杀活动中商品的数量有限；商品的价格会远远低于原来的价格，也就是说，在秒杀活动中，商品会以远远低于原来的价格出售。

例如，秒杀活动的时间仅限于某天上午10点到10点半，商品数量只有10万件，售完为止，而且商品的价格非常低，例如：1元购等业务场景。

限时、限量和限价可以单独存在，也可以组合存在。

（2）活动预热

需要提前配置活动；活动还未开始时，用户可以查看活动的相关信息；秒杀活动开始前，对活动进行大力宣传。

（3）持续时间短

购买的人数数量庞大；商品会迅速售完。

在系统流量呈现上，就会出现一个突刺现象，此时的并发访问量是非常高的，大部分秒杀场景下，商品会在极短的时间内售完。

秒杀三阶段

通常，从秒杀开始到结束，往往会经历三个阶段：

- **准备阶段**：这个阶段也叫作系统预热阶段，此时会提前预热秒杀系统的业务数据，往往这个时候，用户会不断刷新秒杀页面，来查看秒杀活动是否已经开始。在一定程度上，通过用户不断刷新页面的操作，可以将一些数据存储到Redis中进行预热。
- **秒杀阶段**：这个阶段主要是秒杀活动的过程，会产生瞬时的高并发流量，对系统资源会造成巨大的冲击，所以，在秒杀阶段一定要做好系统防护。
- **结算阶段：** 完成秒杀后的数据处理工作，比如数据的一致性问题处理，异常情况处理，商品的回仓处理等。

Redis助力秒杀系统

我们可以在Redis中设计一个Hash数据结构，来支持商品库存的扣减操作，如下所示。

```
seckill:goodsStock:${goodsId}{
    totalCount:200,
    initStatus:0,
    seckillCount:0
}
```

在我们设计的Hash数据结构中，有三个非常主要的属性。

- totalCount：表示参与秒杀的商品的总数量，在秒杀活动开始前，我们就需要提前将此值加载到Redis缓存中。
- initStatus：我们把这个值设计成一个布尔值。秒杀开始前，这个值为0，表示秒杀未开始。可以通过定时任务或者后台操作，将此值修改为1，则表示秒杀开始。
- seckillCount：表示秒杀的商品数量，在秒杀过程中，此值的上限为totalCount，当此值达到totalCount时，表示商品已经秒杀完毕。

我们可以通过下面的代码片段在秒杀预热阶段，将要参与秒杀的商品数据加载的缓存。

```
/**
 * @author binghe
 * @description 秒杀前构建商品缓存代码示例
 */
public class SeckillCacheBuilder{
    private static final String GOODS_CACHE = "seckill:goodsStock:";
    private String getCacheKey(String id) {
        return  GOODS_CACHE.concat(id);
    }
    public void prepare(String id, int totalCount) {
        String key = getCacheKey(id);
        Map<String, Integer> goods = new HashMap<>();
        goods.put("totalCount", totalCount);
        goods.put("initStatus", 0);
        goods.put("seckillCount", 0);
        redisTemplate.opsForHash().putAll(key, goods);
     }
}
```

秒杀开始的时候，我们需要在代码中首先判断缓存中的seckillCount值是否小于totalCount值，如果seckillCount值确实小于totalCount值，我们才能够对库存进行锁定。在我们的程序中，这两步其实并不是原子性的。如果在分布式环境中，我们通过多台机器同时操作Redis缓存，就会发生同步问题，进而引起“超卖”的严重后果。

在电商领域，有一个专业名词叫作“超卖”。顾名思义：“超卖”就是说卖出的商品数量比商品的库存数量多，这在电商领域是一个非常严重的问题。那么，我们如何解决“超卖”问题呢？

Lua脚本完美解决超卖问题

我们如何解决多台机器同时操作Redis出现的同步问题呢？一个比较好的方案就是使用Lua脚本。我们可以使用Lua脚本将Redis中扣减库存的操作封装成一个原子操作，这样就能够保证操作的原子性，从而解决高并发环境下的同步问题。

例如，我们可以编写如下的Lua脚本代码，来执行Redis中的库存扣减操作。

```
local resultFlag = "0"
local n = tonumber(ARGV[1])
local key = KEYS[1]
local goodsInfo = redis.call("HMGET",key,"totalCount","seckillCount")
local total = tonumber(goodsInfo[1])
local alloc = tonumber(goodsInfo[2])
if not total then
    return resultFlag
end
if total >= alloc + n  then
    local ret = redis.call("HINCRBY",key,"seckillCount",n)
    return tostring(ret)
end
return resultFlag
```

我们可以使用如下的Java代码来调用上述Lua脚本。

```
public int secKill(String id, int number) {
    String key = getCacheKey(id);
    Object seckillCount =  redisTemplate.execute(script, Arrays.asList(key), String.valueOf(number));
    return Integer.valueOf(seckillCount.toString());
}
```

这样，我们在执行秒杀活动时，就能够保证操作的原子性，从而有效的避免数据的同步问题，进而有效的解决了“超卖”问题。

一文搞懂PV、UV、VV、IP及其关系与计算

写在前面

> 十一长假基本上过去了，很多小伙伴在假期当中还是保持着持续学习的心态，也有不少小伙伴在微信上问我，让我推送相关的文章。这个时候，我都是抽空来整理小伙伴们的问题，然后，按照顺序进行推文。
>
> PS：这个假期我是哪里也没去，除了在家带娃，就是抽空写文了。有人说：假期没人看技术文？我不信！

小伙伴的疑问

我们还是以一张图来看下小伙伴的疑问吧。

11:50

大佬，在吗？

在，你好

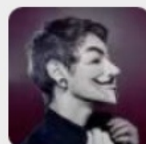
有个问题想请教下

说吧

12:06

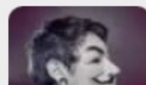
就是我对 PV、UV、VV、IP 表示啥意思不太明白，您可以推一篇文章吗？😁😁

12:09



可以，不过可能要 10 月 8 号推出了

可以可以，谢谢大佬



不客气

接下来，我们就进入正文吧。

什么是PV？

PV即Page View，网站浏览量，指页面浏览的次数，用以衡量网站用户访问的网页数量。

用户每次打开一个页面便记录1次PV，多次打开同一页面则浏览量累计。一般来说，PV与来访者的数量成正比，但是PV并不直接决定页面的真实来访者数量，如同一个来访者通过不断的刷新页面，也可以制造出非常高的PV。

具体的说，PV值就是所有访问者在24小时（0点到24点）内看了某个网站多少个页面或某个网页多少次。PV是指页面刷新的次数，每一次页面刷新，就算做一次PV流量。

度量方法就是从浏览器发出一个对网络服务器的请求（Request），网络服务器接到这个请求后，会将该请求对应的一个网页（Page）发送给浏览器，从而产生了一个PV。那么在这里只要是这个请求发送给了浏览器，无论这个页面是否完全打开（下载完成），那么都是应当计为1个PV。

什么是UV？

UV即Unique Visitor，独立访客数，指一天内访问某站点的人数，以cookie或者Token为依据。

1天内同一访客的多次访问只记录为一个访客。通过IP和cookie是判断UV值的两种方式。

用Cookie分析UV值

当客户端第一次访问某个网站服务器的时候，网站服务器会给这个客户端的电脑发出一个Cookie，通常放在这个客户端电脑的C盘当中。在这个Cookie中会分配一个独一无二的编号，这其中会记录一些访问服务器的信息，如访问时间，访问了哪些页面等等。当你下次再访问这个服务器的时候，服务器就可以直接从你的电脑中找到上一次放进去的Cookie文件，并且对其进行一些更新，但那个独一无二的编号是不会变的。

什么是VV？

VV即Visit View，访客访问的次数，用以记录所有访客一天内访问网站的次数。

当访客完成所有的浏览并最终关掉该网站的所有页面时，便完成了一次访问，同一访客一天内可能有多次访问行为，访问次数累计。

什么是IP？

IP即独立IP数，指一天内使用不同IP地址的用户访问网站的次数，同一IP无论访问了几个页面，独立的IP数均为1。

这里需要注意的是：如果两台机器访问服务器而使用的是同一个IP，那么只能算是一个IP的访问。

IP和UV之间的数据不会有太大的差异，通常UV量和比IP量高出一点，每个UV相对于每个IP更准确地对应一个实际的浏览者。

①UV大于IP

这种情况就是在网吧、学校、公司等，公用相同IP的场所中不同的用户，或者多种不同浏览器访问网站，那么UV数会大于IP数。

②UV小于IP

一般的家庭网络中，大多数电脑使用ADSL拨号上网，所以同一个用户在家里不同时间访问网站时，IP可能会不同，因为它会根据时间变动IP，即动态的IP地址，但是实际访客数唯一，便会出现UV数小于IP数的情况。

实例说明

例如，在家用ADSL拨号上网，早上8点访问了www.binghe.com下的2个页面，下午2点又访问了www.binghe.com下的3个页面。那么，对于www.binghe.com来说，今天的PV、UV、VV、IP各项指标该如何计算？

计算PV

PV指浏览量，PV数等于上午浏览的2个页面和下午浏览的3个页面之和，即 PV = 2 + 3。

计算UV

UV指独立访客数，一天内同一访客的多次访问只计为1个UV，即 UV = 1。

计算VV

VV指访客的访问次数，上午和下午分别有一次访问行为，即 VV = 2。

计算IP

IP为独立IP数，由于ADSL拨号上网每次都IP不同，即 IP = 2。

好了，今天我们就到这儿吧，下期见！！

优化加锁方式时竟然死锁了！！

写在前面

> 今天，在优化程序的加锁方式时，竟然出现了死锁！！到底是为什么呢？！经过仔细的分析之后，终于找到了原因。

为何需要优化加锁方式?

在《【高并发】高并发环境下诡异的加锁问题（你加的锁未必安全）》一文中，我们在转账类TansferAccount中使用TansferAccount.class对象对程序加锁，如下所示。

```
public class TansferAccount{
    private Integer balance;
    public void transfer(TansferAccount target, Integer transferMoney){
        synchronized(TansferAccount.class){
            if(this.balance >= transferMoney){
                this.balance -= transferMoney;
                target.balance += transferMoney;
            }
        }
    }
}
```

这种方式确实解决了转账操作的并发问题，**但是这种方式在高并发环境下真的可取吗?** 试想，如果我们在高并发环境下使用上述代码来处理转账操作，因为TansferAccount.class对象是JVM在加载TansferAccount类的时候创建的，所有的TansferAccount实例对象都会共享一个TansferAccount.class对象。也就是说，**所有TansferAccount实例对象执行transfer()方法时，都是互斥的！！** 换句话说，**所有的转账操作都是串行的！！**

如果所有的转账操作都是串行执行的话，造成的后果就是：账户A为账户B转账完成后，才能进行账户C为账户D的转账操作。如果全世界的网民一起执行转账操作的话，这些转账操作都串行执行，那么，程序的性能是完全无法接受的！！！

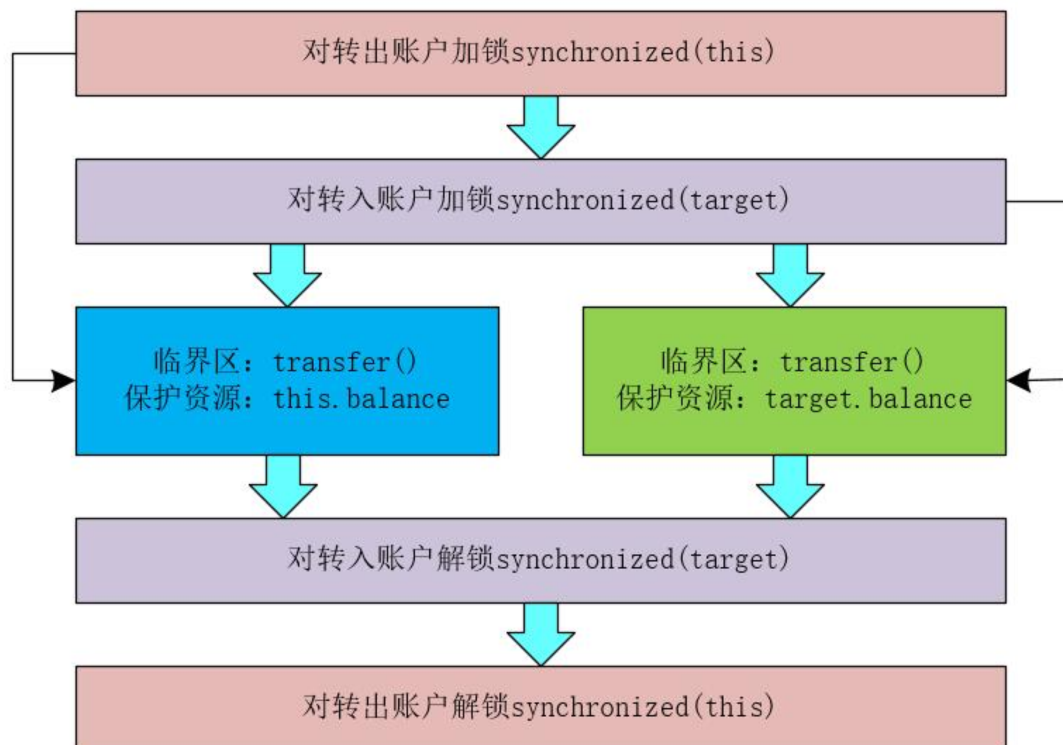
其实，**账户A为账户B转账的操作和账户C为账户D转账的操作完全可以并行执行。**所以，我们必须优化加锁方式，提升程序的性能！！

初步优化加锁方式

既然直接TansferAccount.class对程序加锁在高并发环境下不可取，那么，我们到底应该怎么做呢？！

仔细分析下上面的代码业务，上述代码的转账操作中，涉及到转出账户this和转入账户target，所以，我们可以分别对转出账户this和转入账户target加锁，只有两个账户加锁都成功时，才执行转账操作。这样就能够做到**账户A为账户B转账的操作和账户C为账户D转账的操作完全可以并行执行。**

我们可以将优化后的逻辑用下图表示。



根据上面的分析，我们可以将TansferAccount的代码优化成如下所示。

```
public class TansferAccount{
    //账户的余额
    private Integer balance;
    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        //对转出账户加锁
        synchronized(this){
            //对转入账户加锁
            synchronized(target){
                if(this.balance >= transferMoney){
                    this.balance -= transferMoney;
                    target.balance += transferMoney;
                }
            }
        }
    }
}
```

此时，上面的代码看上去没啥问题，**但真的是这样吗？** 我也希望程序是完美的，但是往往却不是我们想的那样啊！没错，上面的程序会出现 **死锁，** 为什么会出现死锁啊？ 接下来，我们就开始分析一波。

死锁的问题分析

TansferAccount类中的代码看上去比较完美，但是优化后的加锁方式竟然会导致死锁！！！这是我亲测得出的结论！！

关于死锁我们可以结合改进的TansferAccount类举一个简单的场景：假设有线程A和线程B两个线程同时运行在两个不同的CPU上，线程A执行账户A向账户B转账的操作，线程B执行账户B向账户A转账的操作。当线程A和线程B执行到 synchronized(this)代码时，线程A获得了账户A的锁，线程B获得了账户B的锁。当执行到synchronized(target)代码时，线程A尝试获得账户B的锁时，发现账户B已经被线程B锁定，此时线程A开始等待线程B释放账户B的锁；而线程B尝试获得账户A的锁时，发现账户A已经被线程A锁定，此时线程B开始等待线程A释放账户A的锁。

这样，线程A持有账户A的锁并等待线程B释放账户B的锁，线程B持有账户B的锁并等待线程A释放账户A的锁，死锁发生了！！

死锁的必要条件

在如何解决死锁之前，我们先来看下发生死锁时有哪些必要的条件。如果要发生死锁，则必须存在以下四个必要条件，四者缺一不可。

- **互斥条件**

在一段时间内某资源仅为一个线程所占有。此时若有其他线程请求该资源，则请求线程只能等待。

- ## 不可剥夺条件

线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走，即只能由获得该资源的线程自己来释放（只能是主动释放)。

- ## 请求与保持条件

线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时请求线程被阻塞，但对自己已获得的资源保持不放。

- ## 循环等待条件

既然死锁的发生必须存在上述四个条件，那么，大家是不是就能够想到如何预防死锁了呢?

死锁的预防

并发编程中，一旦发生了死锁的现象，则基本没有特别好的解决方法，一般情况下只能重启应用来解决。因此，**解决死锁的最好方法就是预防死锁。**

发生死锁时，必然会存在死锁的四个必要条件。也就是说，如果我们在写程序时，只要"破坏"死锁的四个必要条件中的一个，就能够避免死锁的发生。接下来，我们就一起来探讨下如何"破坏"这四个必要条件。

- ## 破坏互斥条件

互斥条件是我们没办法破坏的，因为我们使用锁为的就是线程之间的互斥。这一点需要特别注意！！！！

- ## 破坏不可剥夺条件

破坏不可剥夺的条件的核心就是让当前线程自己主动释放占有的资源，关于这一点，synchronized是做不到的，我们可以使用java.util.concurrent包下的Lock来解决。此时，我们需要将TansferAccount类的代码修改成类似如下所示。

```
public class TansferAccount{
    private Lock thisLock = new ReentrantLock();
    private Lock targetLock = new ReentrantLock();
    //账户的余额
    private Integer balance;
    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        boolean isThisLock = thisLock.tryLock();
        if(isThisLock){
            try{
                boolean isTargetLock = targetLock.tryLock();
                if(isTargetLock){
                    try{
                         if(this.balance >= transferMoney){
                            this.balance -= transferMoney;
                            target.balance += transferMoney;
                        }
                    }finally{
                        targetLock.unlock
                    }
                }
            }finally{
                thisLock.unlock();
            }
        }
    }
}
```

其中Lock中有两个tryLock方法，分别如下所示。

- tryLock()方法

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

- tryLock(long time, TimeUnit unit)方法

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

- ## 破坏请求与保持条件

破坏请求与保持条件，我们可以一次性申请所需要的所有资源，例如在我们完成转账操作的过程中，我们一次性申请账户A和账户B，两个账户都申请成功后，再执行转账的操作。此时，我们需要再创建一个申请资源的类ResourcesRequester，这个类的作用就是申请资源和释放资源。同时，TansferAccount类中需要持有一个ResourcesRequester类的单例对象，当我们需要执行转账操作时，首先向ResourcesRequester同时申请转出账户和转入账户两个资源，申请成功后，再锁定两个资源；当转账操作完成后，释放锁并释放ResourcesRequester类申请的转出账户和转入账户资源。

ResourcesRequester类的代码如下所示。

```
public class ResourcesRequester{
    //存放申请资源的集合
    private List<Object> resources = new ArrayList<Object>();
    //一次申请所有的资源
    public synchronized boolean applyResources(Object source, Object target){
        if(resources.contains(source) || resources.contains(target)){
            return false;
        }
        resources.add(source);
        resources.add(targer);
        return true;
    }

    //释放资源
    public synchronized void releaseResources(Object source, Object target){
        resources.remove(source);
        resources.remove(target);
    }
}
```

此时，TansferAccount类的代码如下所示。

```
public class TansferAccount{
    //账户的余额
    private Integer balance;
    //ResourcesRequester类的单例对象
    private ResourcesRequester requester;

    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        //自旋申请转出账户和转入账户，直到成功
        while(!requester.applyResources(this, target)){
            //循环体为空
            ;
        }
        try{
            //对转出账户加锁
            synchronized(this){
                //对转入账户加锁
                synchronized(target){
                    if(this.balance >= transferMoney){
                        this.balance -= transferMoney;
                        target.balance += transferMoney;
                    }
                }
            }
        }finally{
            //最后释放账户资源
            requester.releaseResources(this, target);
        }

    }
}
```

- ## 破坏循环等待条件

破坏循环等待条件，则可以通过对资源排序，按照一定的顺序来申请资源，然后按照顺序来锁定资源，可以有效的避免死锁。

例如，在我们的转账操作中，往往每个账户都会有一个唯一的id值，我们在锁定账户资源时，可以按照id值从小到大的顺序来申请账户资源，并按照id从小到大的顺序来锁定账户，此时，程序就不会再进行循环等待了。

程序代码如下所示。

```
public class TansferAccount{
    //账户的id
    private Integer id;
    //账户的余额
    private Integer balance;
    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        TansferAccount beforeAccount = this;
        TansferAccount afterAccount = target;
        if(this.id > target.id){
            beforeAccount = target;
            afterAccount = this;
        }
        //对转出账户加锁
        synchronized(beforeAccount){
            //对转入账户加锁
            synchronized(afterAccount){
                if(this.balance >= transferMoney){
                    this.balance -= transferMoney;
                    target.balance += transferMoney;
                }
            }
        }
    }
}
```

总结

在并发编程中，使用细粒度锁来锁定多个资源时，要时刻注意死锁的问题。另外，避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的线程申请资源必须以一定的顺序来操作进而避免死锁。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

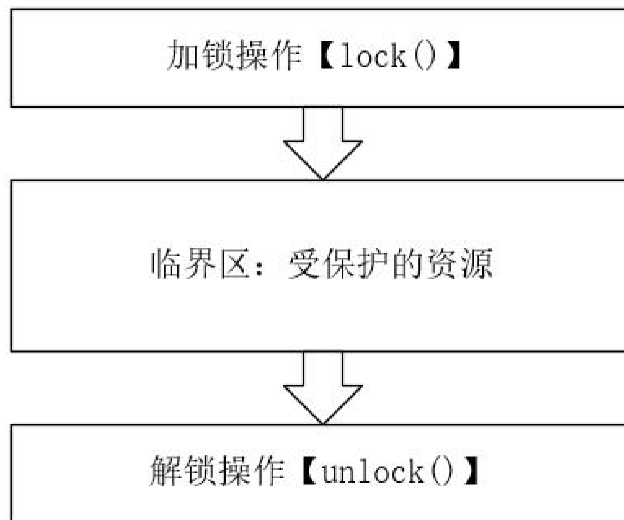
如何使用互斥锁解决多线程的原子性问题

如何保证原子性

那么，如何解决线程切换带来的原子性问题呢？答案是**保证多线程之间的互斥性。也就是说，在同一时刻只有一个线程在执行！**如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，都能保证多线程之间的原子性了。

锁模型

说到线程之间的互斥，我们可以想到在并发编程中使用锁来保证线程之前的互斥性。我们可以将使用锁的模型简单的使用下图来表示。

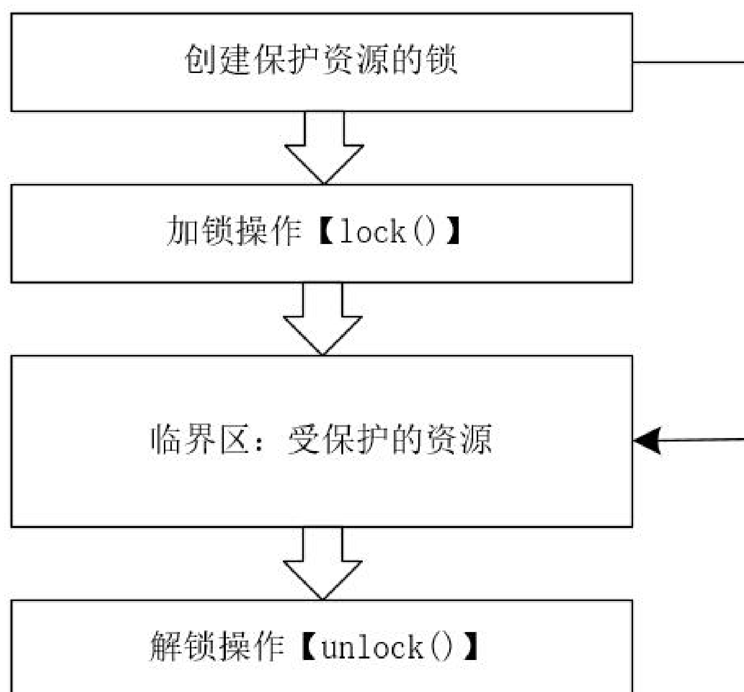


我们可以将上图中受保护的资源，也就是需要多线程之间互斥执行的代码称为临界区。线程进入临界区之前，会首先尝试加锁操作lock()，如果加锁成功，则进入临界区执行临界区中的代码，则当前线程持有锁；如果加锁失败，就会等待，直到持有锁的线程释放锁后，当前线程获取到锁进入临界区；进入临界区的线程执行完代码后，会执行解锁操作unlock()。

其实，在这个锁模型中，我们忽略了一些非常重要的内容：那就是我们对什么东西加了锁？需要我们保护的资源又是什么呢？

改进的锁模型

在并发编程中对资源进行加锁操作时，我们需要明确对什么东西加了锁？而需要我们保护的资源又是什么？只有明确了这两点，才能更好的利用Java中的互斥锁。所以，我们需要将锁模型进行修改，修改后的锁模型如下图所示。



在改进的锁模型中，首先创建一把保护资源的锁，使用这个保护资源的锁进行加锁操作，然后进入临界区执行代码，最后进行解锁操作释放锁。其中，创建的保护资源的锁，就是对临界区特定的资源进行保护。

这里需要注意的是：我们在改进的锁模型中，特意将创建保护资源的锁用箭头指向了临界区中的受保护的资源。目的是为了说明特定资源的锁是为了保护特定的资源，如果一个资源的锁保护了其他的资源，那么就会出现诡异的Bug问题，这样的Bug非常不好调试，因为我们自身会觉得，我明明已经对代码进行了加锁操作，可为什么还会出现问题呢？如果出现了这种问题，你就要排查下你创建的锁，是不是真正要保护你需要保护的资源了。

Java中的synchronized锁

说起，Java中的synchronized锁，相信大家并不陌生了，synchronized关键字可以用来修饰方法，也可以用来修饰代码块。例如，下面的代码片段所示。

```
public class LockTest{
    //使用synchronized修饰非静态方法
```

```
    punlic synchronized void execute(){
        //临界区：受保护的资源
    }

    //使用synchronized修饰静态方法
    public synchronized static void submit(){
        //临界区：受保护的资源
    }

    //创建需要加锁的对象
    private Object obj = new Object();
    //修饰代码块
    public void run(){
        synchronized(obj){
            //临界区：受保护的资源
        }
    }
}
```

在上述的代码中，我们只是对方法（包括静态方法和非静态方法）和代码块使用了synchronized关键字，并没有执行lock()和unlock()操作。本质上，synchronized的加锁和解锁操作都是由JVM来完成的，Java编译器会在synchronized修饰的方法或代码块的前面自动加上加锁操作，而在其后面自动加上解锁操作。

在使用synchronized关键字加锁时，Java规定了一些隐式的加锁规则。

- 当使用synchronized关键字修饰静态方法时，锁定的是当前类的Class对象。
- 当使用synchronized关键字修饰非静态方法时，锁定的是当前实例对象this。
- 当使用synchronized关键字修饰代码块时，锁定的是实际传入的对象。

再次深究count+=1的问题

如果多个线程并发的对共享变量count执行加1操作，就会出现问题。此时，我们可以使用synchronized锁来尝试解决下这个问题。

例如，TestCount类中有两个方法，一个是getCount()方法，用来获取count的值；另一个是incrementCount()方法，用来给count值加1，并且incrementCount()方法使用synchronized关键字修饰，如下所示。

```
public class TestCount{
    private long count = 0L;
    public long getCount(){
        return count;
    }
    public synchronized void incrementCount(){
        count += 1;
    }
}
```

通过上面的代码，我们肯定的是incrementCount()方法被synchronized关键字修饰后，无论是单核CPU还是多核CPU，此时只有一个线程能够执行incrementCount()方法，所以，incrementCount()方法一定可以保证原子性。

这里，我们还要思考另一个问题：上面的代码是否存在可见性问题呢? 回答这个问题之间，我们还需要看下Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。**

在上面的代码中，使用synchronized关键字修饰的incrementCount()方法是互斥的，也就是说，在同一时刻只有一个线程执行incrementCount()方法中的代码；而Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。**指的是前一个线程的解锁操作对后一个线程的加锁操作可见，再综合Happens-Before原则的【原则三】传递规则：**如果A Happens-Before B，并且B Happens-Before C，则A Happens-Before C。**我们可以得出一个结论：**前一个线程在临界区修改的共享变量（该操作在解锁之前），对后面进入这个临界区（该操作在加锁之后）的线程是可见的。**

经过上面的分析，如果多个线程同时执行incrementCount()方法，是可以保证可见性的，也就是说，如果有100个线程同时执行incrementCount()方法，count变量的最终结果为100。

但是，还没完，TestCount类中还有一个getCount()方法，如果执行了incrementCount()方法，count变量的值对getCount()方法是可见的吗?

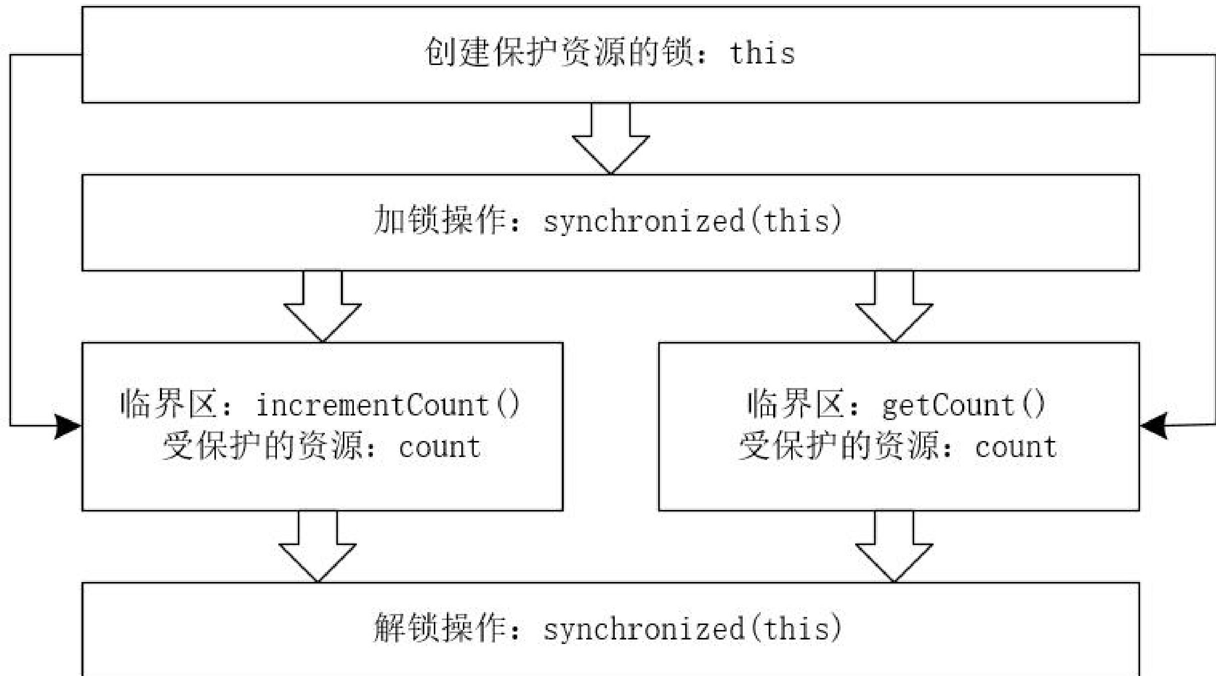
在Happens-Before原则的【原则四】锁定规则：**对一个锁的解锁操作 Happens-Before于后续对这个锁的加锁操作。**只能保证后续对这个锁的加锁的可见性。而getCount()方法没有执行加锁操作，所以，无法保证incrementCount()方法的执行结果对getCount()方法可见。

如果需要保证incrementCount()方法的执行结果对getCount()方法可见，我们也需要为getCount()方法使用synchronized关键字修饰。所以，TestCount类的代码如下所示。

```
public class TestCount{
    private long count = 0L;
    public synchronized long getCount(){
        return count;
    }
    public synchronized void incrementCount(){
        count += 1;
    }
}
```

此时，为getCount()方法也添加了synchronized锁，而且getCount()方法和incrementCount()方法锁定的都是this对象，线程进入getCount()方法和incrementCount()方法时，必须先获得this这把锁，所以，getCount()方法和incrementCount()方法是互斥的。也就是说，此时，incrementCount()方法的执行结果对getCount()方法可见。

我们也可以简单的使用下图来表示这个互斥的逻辑。



修改测试用例

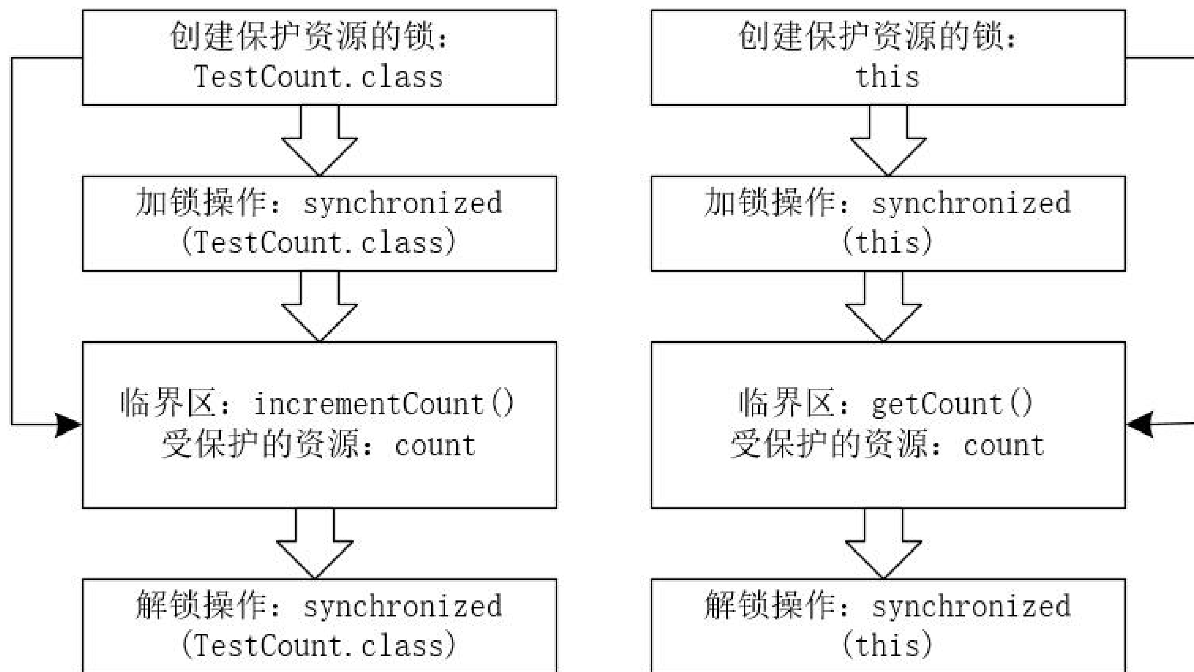
我们将上面的测试代码稍作修改，将count的修改为静态变量，将incrementCount()方法修改为静态方法。此时的代码如下所示。

```
public class TestCount{
    private static long count = 0L;
    public synchronized long getCount(){
        return count;
    }
    public synchronized static void incrementCount(){
        count += 1;
    }
}
```

那么，问题来了，getCount()方法和incrementCount()方法是否存在并发问题呢？

接下来，我们一起分析下这段代码：其实这段代码中是在用两个不同的锁来保护同一个资源count，两个锁分别为this对象和TestCount.class对象。也就是说，getCount()方法和incrementCount()方法获取的是两个不同的锁，二者的临界区没有互斥关系，incrementCount()方法对count变量的修改无法保证对getCount()方法的可见性。所以，**修改后的代码会存在并发问题。**

我们也可以使用下图来简单的表示这个逻辑。



总结

保证多线程之间的互斥性。也就是说，在同一时刻只有一个线程在执行！如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，都能保证多线程之间的原子性了。

注意：在Java中，也可以使用Lock锁来实现多线程之间的互斥，大家可以自行使用Lock锁实现。

高并发环境下诡异的加锁问题（你加的锁未必安全）

声明

特此声明：文中有关支付宝账户的说明，只是用来举例，实际支付宝账户要比文中描述的复杂的多。也与文中描述的完全不同。

前言

> 很多网友留言说：在编写多线程并发程序时，我明明对共享资源加锁了啊？为什么还是出问题呢？问题到底出在哪里呢？其实，我想说的是：你的加锁姿势正确吗？你真的会使用锁吗？错误的加锁方式不但不能解决并发问题，而且还会带来各种诡异的Bug问题，有时难以复现！

我们知道在并发编程中，不能使用多把锁保护同一个资源，因为这样达不到线程互斥的效果，存在线程安全的问题。相反，却可以使用同一把锁保护多个资源。那么，如何使用同一把锁保护多个资源呢？又如何判断我们对程序加的锁到底是不是安全的呢？我们就一起来深入探讨这些问题！

分析场景

我们在分析多线程中如何使用同一把锁保护多个资源时，可以将其结合具体的业务场景来看，比如：需要保护的多个资源之间有没有直接的业务关系。如果需要保护的资源之间没有直接的业务关系，那么如何对其加锁；如果有直接的业务关系，那么如何对其加锁？接下来，我们就顺着这两个方向进行深入说明。

没有直接业务关系的场景

例如，我们的支付宝账户，有针对余额的付款操作，也有针对账户密码的修改操作。**本质上，这两种操作之间没有直接的业务关系，此时，我们可以为账户的余额和账户密码分配不同的锁来解决并发问题。**

例如，在支付宝账户AlipayAccount类中，有两个成员变量，分别是账户的余额balance和账户的密码password。付款操作的pay()方法和查看余额操作的getBalance()方法会访问账户中的成员变量balance，对此，我们可以创建一个balanceLock锁对象来保护balance资源；另外，更改密码操作的updatePassword()方法和查看密码的getPassowrd()方法会访问账户中的成员变量password，对此，我们可以创建一个passwordLock锁对象来保护password资源。

具体的代码如下所示。

```
public class AlipayAccount{
```

```
    //保护balance资源的锁对象
    private final Object balanceLock = new Object();
    //保护password资源的锁对象
    private final Object passwordLock = new Object();
    //账户余额
    private Integer balance;
    //账户的密码
    private String password;

    //支付方法
    public void pay(Integer money){
        synchronized(balanceLock){
            if(this.balance >= money){
                this.balance -= money;
            }
        }
    }
    //查看账户中的余额
    public Integer getBalance(){
        synchronized(balanceLock){
            return this.balance;
        }
    }

    //修改账户的密码
    public void updatePassword(String password){
        synchronized(passwordLock){
            this.password = password;
        }
    }

    //查看账户的密码
    public String getPassword(){
        synchronized(passwordLock){
            return this.password;
        }
    }
}
```

这里，我们也可以使用一把互斥锁来保护balance资源和password资源，例如都使用balanceLock锁对象，也可以都使用passwordLock锁对象，甚至也都可以使用this对象或者干脆每个方法前加一个synchronized关键字。

但是，如果都使用同一个锁对象的话，那么，程序的性能就太差了。会导致没有直接业务关系的各种操作都串行执行，这就违背了我们并发编程的初衷。实际上，我们使用两个锁对象分别保护balance资源和password资源，付款和修改账户密码是可以并行的。

存在直接业务关系的场景

例如，我们使用支付宝进行转账操作。假设账户A给账户B转账100，A账户减少100元，B账户增加100元。两个账户在业务中有直接的业务关系。例如，下面的TansferAccount类，有一个成员变量balance和一个转账的方法transfer()，代码如下所示。

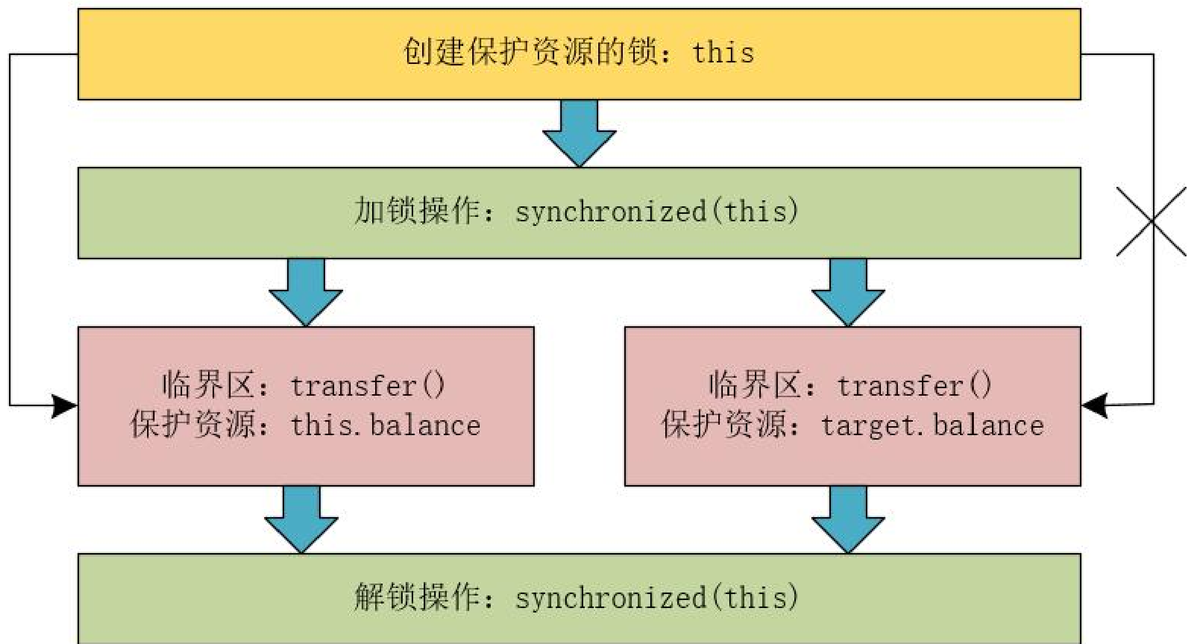
```
public class TansferAccount{
    private Integer balance;
    public void transfer(TansferAccount target, Integer transferMoney){
        if(this.balance >= transferMoney){
            this.balance -= transferMoney;
            target.balance += transferMoney;
        }
    }
}
```

在上面的代码中，如何保证转账操作不会出现并发问题呢？很多时候我们的第一反应就是给transfer()方法加锁，如下代码所示。

```
public class TansferAccount{
    private Integer balance;
    public synchronized void transfer(TansferAccount target, Integer transferMoney){
        if(this.balance >= transferMoney){
            this.balance -= transferMoney;
            target.balance += transferMoney;
        }
    }
}
```

我们仔细分析下，**上面的代码真的是安全的吗？！**其实，在这段代码中，synchronized临界区中存在两个不同的资源，分别是转出账户的余额this.balance和转入账户的余额target.balance，这里只用到了一把锁synchronized(this)。说到这里，大家有没有一种豁然开朗的感觉。没错，**问题就出现在synchronized(this)这把锁上，这把锁只能保护this.balance资源，而无法保护target.balance资源。**

我们可以使用下图来表示这个逻辑。



从上图我们也可以发现，**this锁对象只能保护this.balance资源，而不能保护target.balance资源。**

接下来，我们再看一个场景：假设存在A、B、C三个账户，余额都是200，此时我们使用两个线程分别执行两个转账操作：账户A给账户B转账100，账户B给账户C转账100。理论上，账户A的余额为100，账户B的余额为200，账户C的余额为300。

真的是这样吗？我们假设线程A和线程B同时在两个不同的CPU上执行，线程A执行账户A给账户B转账100的操作，线程B执行账户B给账户C转账100的操作。两个线程之间是互斥的吗？显然不是，按照TansferAccount的代码来看，线程A锁定的是账户A的实例，线程B锁定的是账户B的实例。所以，线程A和线程B能够同时进入transfer()方法。此时，线程A和线程B都能够读取到账户B的余额为200。**两个线程都完成转账操作后，B的账户余额可能为300，也可能为100，但是不可能为200。**

这是为什么呢？线程A和线程B同时读取到账户B的余额为200，如果线程A的转账操作晚于线程B的转账操作对balance的写入，则账户B的余额为300；如果线程A的转账操作早于线程B的转账操作对balance的写入，则账户B的余额为100。无论如何账户B的余额都不会是200。

综上所示，TansferAccount的代码根本无法解决并发问题！

正确的加锁

如果我们希望对转账操作中涉及的多个资源加锁，那我们的锁就必须要覆盖所有需要保护的资源。

在前面的TansferAccount类中，this是对象级别的锁，这就导致了线程A和线程B执行过程中所获取到的锁是不同的，那么如何让两个线程共享同一把锁呢？！

其中，方案有很多，一种简单的方式，就是在TansferAccount类的构造方法中传入一个balanceLock锁对象，以后在创建TansferAccount类对象的时候，每次传入相同的balanceLock锁对象，并在transfer方法中使用balanceLock锁对象加锁即可。这样，所有创建的TansferAccount类对象就会共享balanceLock锁。代码如下所示。

```
public class TansferAccount{
```

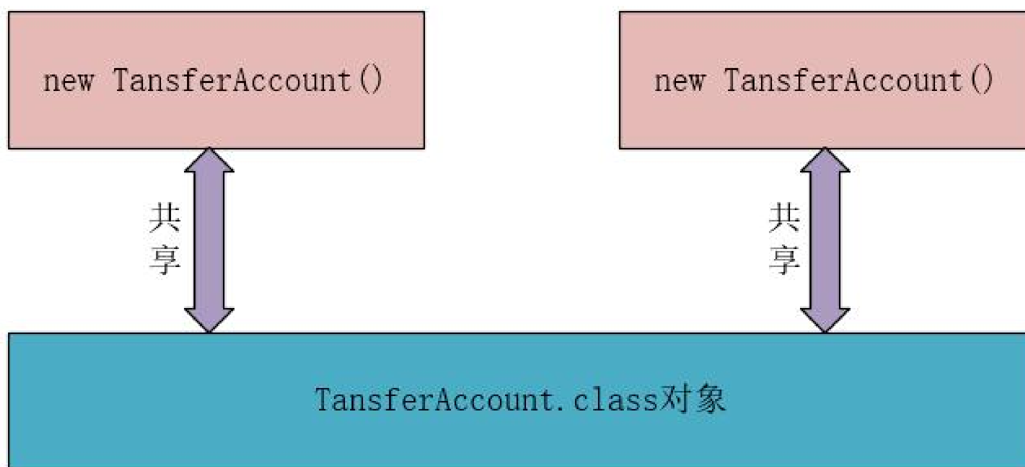
```
    private Integer balance;
    private Object balanceLock;
    private TansferAccount(){}
    public TansferAccount(Object balanceLock){
        this.balanceLock = balanceLock;
    }
    public void transfer(TansferAccount target, Integer transferMoney){
        synchronized(this.balanceLock){
             if(this.balance >= transferMoney){
                this.balance -= transferMoney;
                target.balance += transferMoney;
            }
        }
    }
}
```

那么，问题又来了：这样解决问题真的完美吗？！

上述代码虽然解决了转账操作的并发问题，但是它真的就完美了吗？！仔细分析后，我们发现，并不是想象中的那么完美。因为它要求创建TansferAccount对象的时候，必须传入同一个balanceLock对象，如果传入的不是同一个balanceLock对象，就不能保证并发带来的线程安全问题了！在实际的项目中，创建TansferAccount对象的操作可能被分散在多个不同的项目工程中，这样很难保证传入的balanceLock对象是同一个对象。

所以，在创建TansferAccount对象时传入同一个balanceLock锁对象的方案，虽然能够解决转账的并发问题，但是却无法在实际项目中被有效的采用！

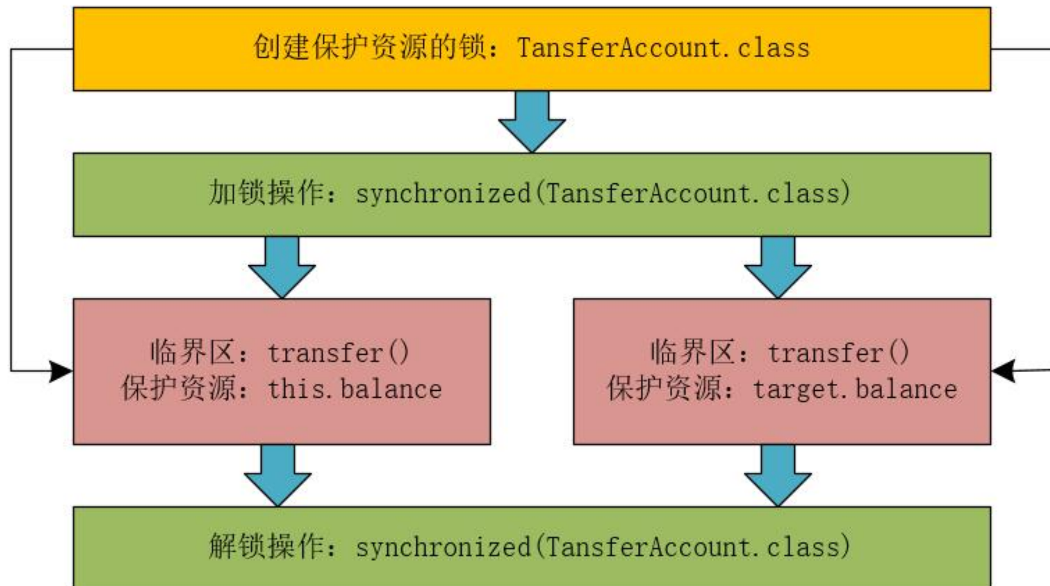
还有没有其他的方案呢？答案是有！别忘了JVM在加锁类的时候，会为类创建一个Class对象，而这个Class对象对于类的实例对象来说是共享的，也就是说，无论创建多少个类的实例对象，这个Class对象都是同一个，这是由JVM来保证的。



说到这里，我们就能够想到使用如下方式对转账操作加锁。

```
public class TansferAccount{
    private Integer balance;
    public void transfer(TansferAccount target, Integer transferMoney){
        synchronized(TansferAccount.class){
            if(this.balance >= transferMoney){
                this.balance -= transferMoney;
                target.balance += transferMoney;
            }
        }
    }
}
```

我们可以使用下图表示这个逻辑。



这样，无论创建多少个TansferAccount对象，都会共享同一把锁，解决了转账的并发问题。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

高并发场景下创建多少线程才合适？一条公式帮你搞定！！

创建多少线程合适， 要看多线程具体的应用场景。 一般来说，我们可以将程序分为：**CPU密集型程序和I/O密集型程序**， 而针对于CPU密集型程序和I/O密集型程序，其计算最佳线程数的方法是不同的 。

CPU密集型程序

对于CPU密集型计算， 多线程本质上是提升多核CPU的利用率， 所以对于一个4核的CPU， 每个核一个线程， 理论上创建4个线程就可以了， 再多创建线程也只是增加线程切换的成本。 所以， 对于CPU密集型的计算场景， 理论上"线程的量=CPU核数"就是最合适的。 但是在实际工作中， 一般会将线程数量设置为"CPU核数+1"， 这样的话， 当线程因为偶尔的内存页失效或其他原因导致阻塞时， 这个额外的线程可以顶上， 从而保证CPU的利用率 。

所以，在CPU密集型的程序中，一般可以将线程数设置为CPU核数+1。

I/O密集型程序

对于I/O密集型的程序，最佳的线程数是与程序中CPU计算和I/O操作的耗时比相关。总体来说，可以将其总结为如下的公式。

单核CPU

最佳线程数 = 1 + （I/O耗时 / CPU耗时）

我们令R=I/O耗时 / CPU耗时， 可以这样理解： 当线程A执行IO操作时， 另外R个线程正好执行完各自的CPU计算。 这样CPU的利用率就达到了100%。

多核CPU

多核CPU的最佳线程数在单核CPU最佳线程数的基础上，乘以CPU核数即可，如下所示。

最佳线程数=CPU核数 * [1 + （I/O耗时 / CPU耗时）]

总结

上述公式计算的结果为最佳理论值，实际工作中还是要通过实际压测数据来找到最佳线程数，将硬件的性能发挥到极致。

终于弄懂为什么局部变量是线程安全的了！！

写在前面

> 相信很多小伙伴都知道局部变量是线程安全的，那你知道为什么局部变量是线程安全的吗？

前言

多个线程同时访问共享变量时，会导致并发问题。那么，如果将变量放在方法内部，是不是还会存在并发问题呢？如果不存在并发问题，那么为什么不会存在并发问题呢？

著名的斐波那契数列

记得上学的时候，我们都会遇到这样一种题目，打印斐波那契数列。斐波那契数列是这样的一个数列：1、1、2、3、5、8、13、21、34...，也就是说第1项和第2项是1，从第3项开始，每一项都等于前2项之和。我们可以使用下面的代码来生成斐波那契数列。

```
//生成斐波那契数列
public int[] fibonacci(int n){
    //存放结果的数组
    int[] result = new int[n];
    //数组的第1项和第2项为1
    result[0] = result[1] = 1;
    //计算第3项到第n项
    for(int i = 2; i < n; i++){
        result[i] = result[i-2] + result[i-1];
    }
    return result;
}
```

假设此时有很多个线程同时调用fibonacci()方法来生成斐波那契数列，**对于方法中的局部变量result，会不会存在线程安全的问题呢？答案是：不会！！**

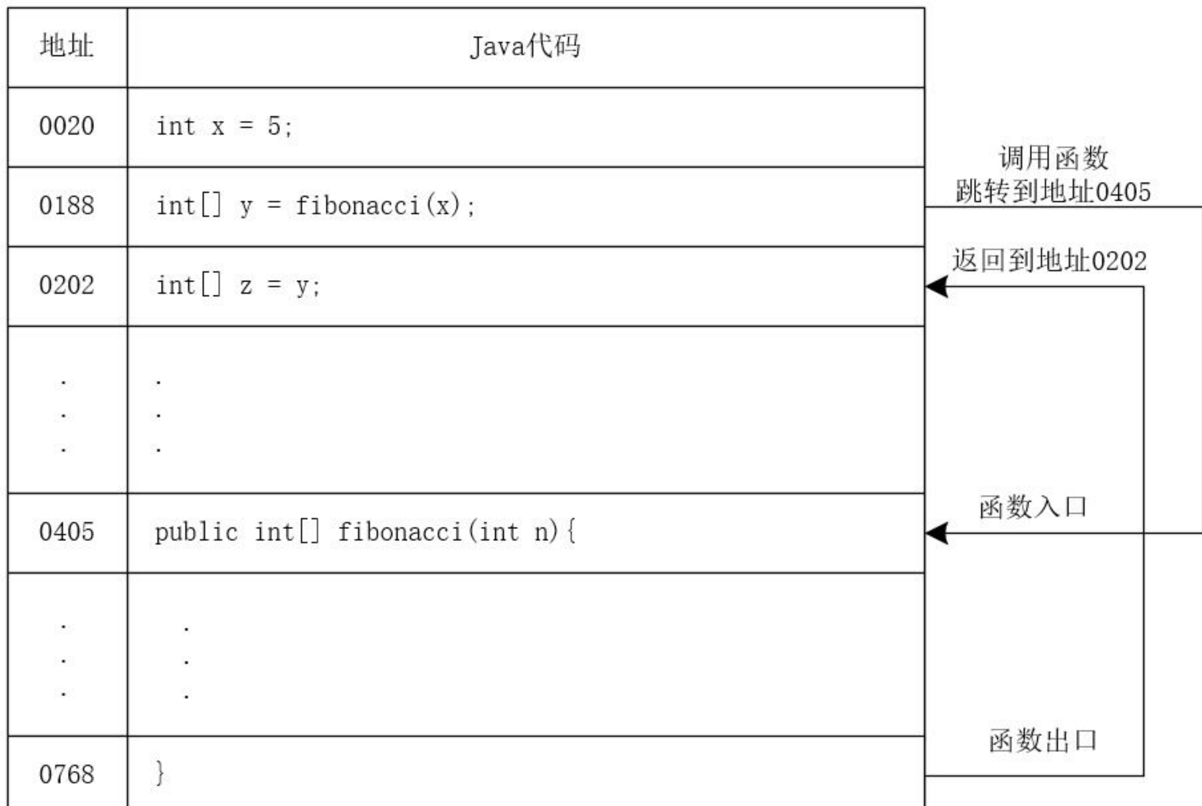
接下来，我们就深入分析下为什么局部变量不会存在线程安全的问题！

方法是如何被执行的?

我们以下面的三行代码为例。

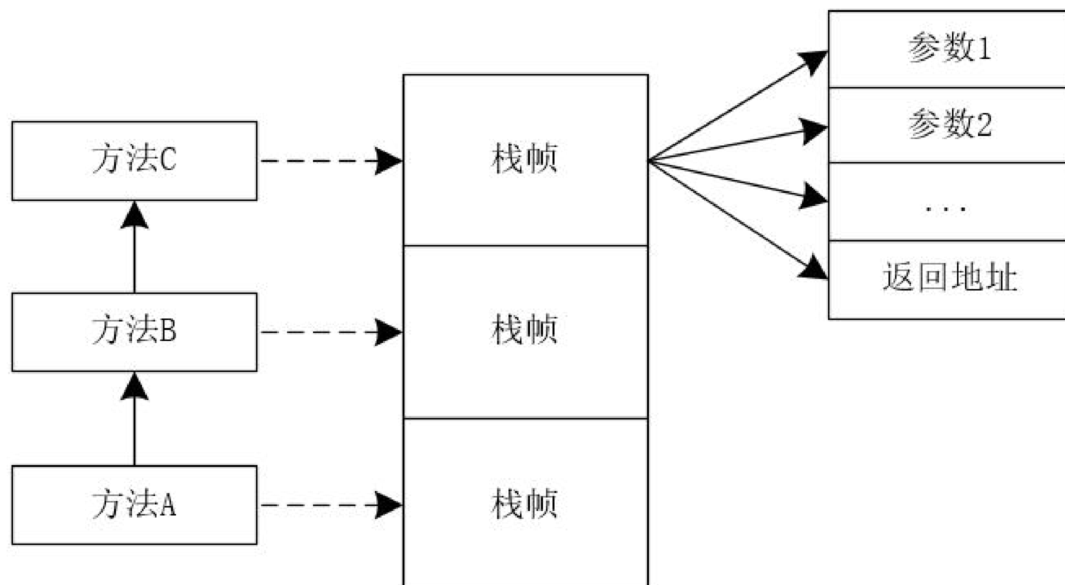
```
int x = 5;
int[] y = fibonacci(x);
int[] z = y;
```

当我们调用fibonacci(x)时，CPU要先找到fibonacci()方法的地址，然后跳转到这个地址去执行代码，执行完毕后，需要返回并找到调用方法的下一条语句的地址，也就是int[] z = y的地址，再跳到这个地址去执行。我们可以将这个过程简化成下图所示。



这里需要注意的是：CPU会通过堆栈寄存器找到调用方法的参数和返回地址。

例如，有三个方法A、B、C，调用关系为A调用B，B调用C。在运行时，会构建出相应的调用栈，我们可以用下图简单的表示这个调用栈。

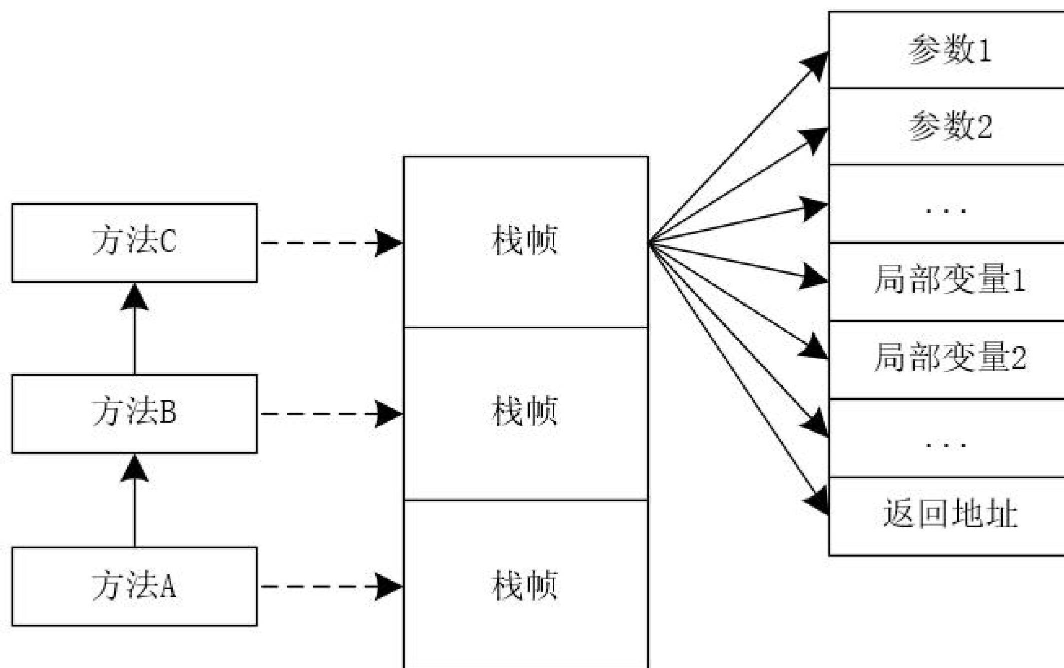


每个方法在调用栈里都会有自己独立的栈帧，每个栈帧里都有对应方法需要的参数和返回地址。当调用方法时，会创建新的栈帧，并压入调用栈；当方法返回时，对应的栈帧就会被自动弹出。

我们可以这样说：**栈帧是在调用方法时创建，方法返回时"消亡"。**

局部变量存放在哪里?

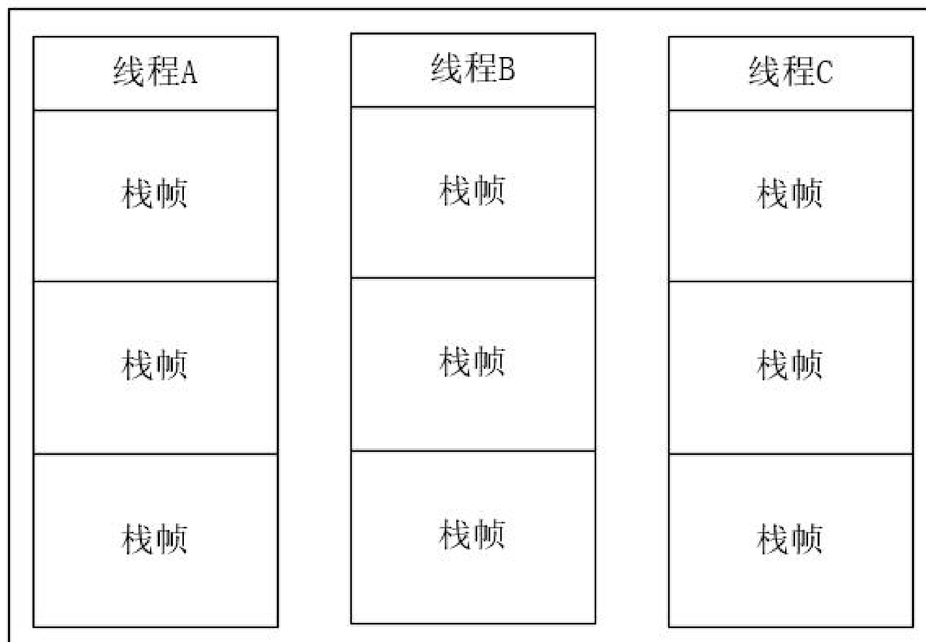
局部变量的作用域在方法内部，当方法执行完，局部变量也就没用了。可以这么说，方法返回时，局部变量也就"消亡"了。此时，我们会联想到调用栈的栈帧。没错，**局部变量就是存放在调用栈里的。**此时，我们可以将方法的调用栈用下图表示。



很多人都知道，局部变量会存放在栈里。**如果一个变量需要跨越方法的边界，就必须创建在堆里。**

调用栈与线程

两个线程就可以同时用不同的参数调用相同的方法。**那么问题来了，调用栈和线程之间是什么关系呢？答案是：每个线程都有自己独立的调用栈。**我们可以使用下图来简单的表示这种关系。



此时，我们在看下文中开头的问题：**Java方法内部的局部变量是否存在并发问题？答案是不存在并发问题！因为每个线程都有自己的调用栈，局部变量保存在线程各自的调用栈里，不会共享，自然也就不存在并发问题。**

线程封闭

方法里的局部变量，因为不会和其他线程共享，所以不会存在并发问题。这种解决问题的技术也叫做线程封闭。官方的解释为：仅在单线程内访问数据。由于不存在共享，所以即使不设置同步，也不会出现并发问题！

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

线程的生命周期其实没有我们想象的那么简单！！

写在前面

> 在【高并发专题】前面的文章中，我们简单介绍了线程的生命周期和线程的几个重要状态，并以代码的形式实现了线程是如何进入各个状态的。今天，我们就结合**操作系统线程和编程语言线程**再次深入探讨线程的生命周期问题，线程的生命周期其实没有我们想象的那么简单！！

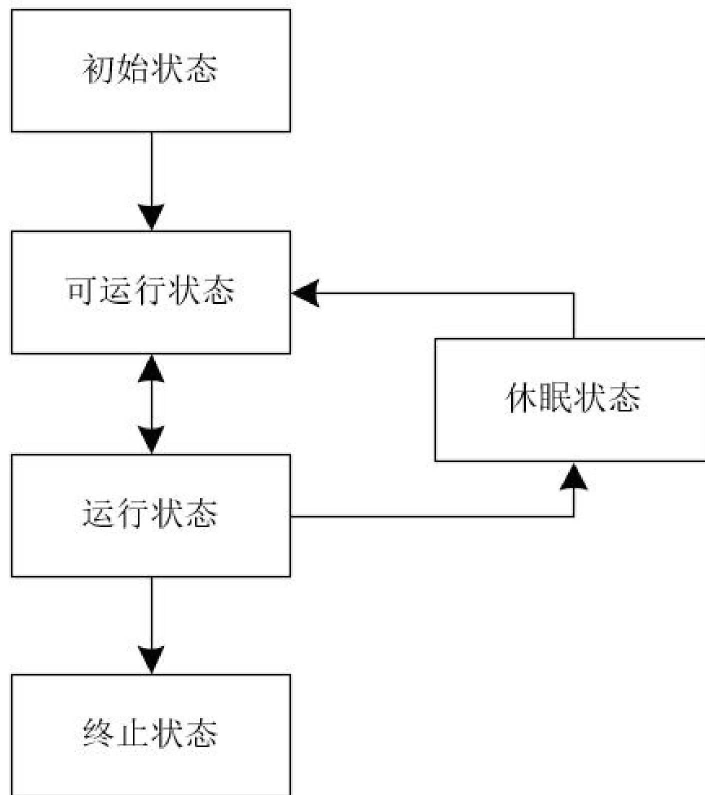
理解线程的生命周期本质上理解了**生命周期中各个节点的状态转换机制**就可以了。

接下来，我们分别就**通用线程生命周期和Java语言的线程生命周期**分别进行详细说明。

通用的线程生命周期

通用的线程生命周期总体上可以分为五个状态：**初始状态、可运行状态、运行状态、休眠状态和终止状态。**

我们可以简单的使用下图来表示这五种状态。



初始状态

线程已经被创建，但是不允许分配CPU执行。**需要注意的是：这个状态属于编程语言特有，这里指的线程已经被创建，仅仅指在编程语言中被创建，在操作系统中，并没有创建真正的线程。**

可运行状态

线程可以分配CPU执行。此时，**操作系统中的线程被成功创建，可以分配CPU执行。**

运行状态

当操作系统中存在空闲的CPU，操作系统会将这个空闲的CPU分配给一个处于可运行状态的线程，被分配到CPU的线程的状态就转换成了运行状态

休眠状态

运行状态的线程调用一个阻塞的API（例如，以阻塞的方式读文件）或者等待某个事件（例如，等待某个条件变量等），线程的状态就会转换到休眠状态。**此时线程会释放CPU资源，休眠状态的线程没有机会获得CPU的使用权。**一旦等待的条件出现，线程就会从休眠状态转换到可运行状态。

终止状态

线程执行完毕或者出现异常就会进入终止状态，终止状态的线程不会切换到其他任何状态，这也意味着**线程的生命周期结束了。**

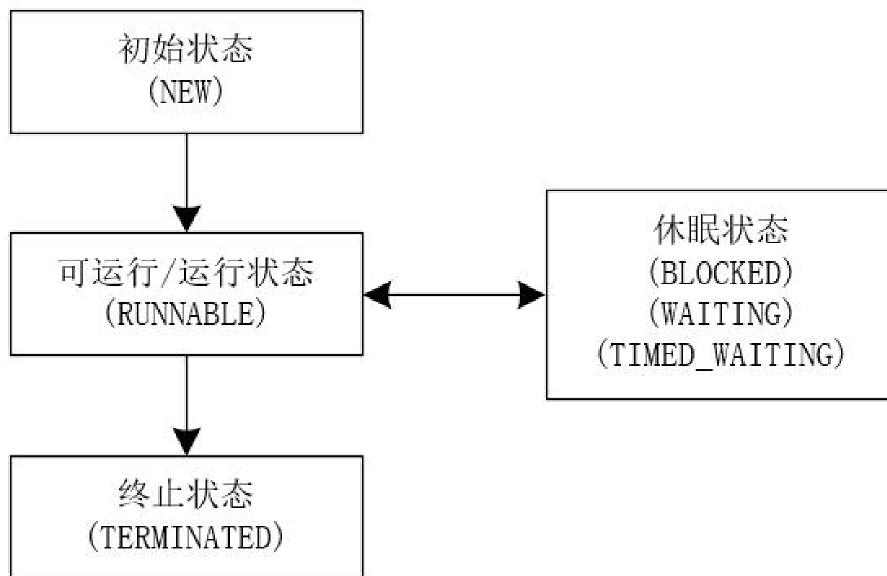
以上就是通用的线程生命周期，下面，我们再看对比看下Java语言中的线程生命周期。

Java中的线程生命周期

Java中的线程生命周期总共可以分为六种状态：**初始化状态（NEW）、可运行/运行状态（RUNNABLE）、阻塞状态（BLOCKED）、无时限等待状态（WAITING）、有时限等待状态（TIMED_WAITING）、终止状态（TERMINATED）。**

需要大家重点理解的是：虽然Java语言中线程的状态比较多，但是，其实在操作系统层面，Java线程中的阻塞状态（BLOCKED）、无时限等待状态（WAITING）、有时限等待状态（TIMED_WAITING）都是一种状态，即通用线程生命周期中的休眠状态。也就是说，只要Java中的线程处于这三种状态时，那么，这个线程就没有CPU的使用权。

理解了这些之后，我们就可以使用下面的图来简单的表示Java中线程的生命周期。



我们也可以这样理解阻塞状态（BLOCKED）、无时限等待状态（WAITING）、有时限等待状态（TIMED_WAITING），它们是导致线程休眠的三种原因！

接下来，我们就看看Java线程中的状态是如何转化的。

RUNNABLE与BLOCKED的状态转换

只有一种场景会触发这种转换，就是线程等待synchronized隐式锁。synchronized修饰的方法、代码块同一时刻只允许一个线程执行，其他的线程则需要等待。此时，等待的线程就会从RUNNABLE状态转换到BLOCKED状态。当等待的线程获得synchronized隐式锁时，就又会从BLOCKED状态转换到RUNNABLE状态。

这里，需要大家注意：线程调用阻塞API时，在操作系统层面，线程会转换到休眠状态。但是在JVM中，Java线程的状态不会发生变化，也就是说，Java线程的状态仍然是RUNNABLE状态。JVM并不关心操作系统调度相关的状态，在JVM角度来看，等待CPU使用权（操作系统中的线程处于可执行状态）和等待IO操作（操作系统中的线程处于休眠状态）没有区别，都是在等待某个资源，所以，将其都归入了RUNNABLE状态。

我们平时所说的Java在调用阻塞API时，线程会阻塞，指的是操作系统线程的状态，并不是Java线程的状态。

RUNNABLE与WAITING状态转换

线程从RUNNABLE状态转换成WAITING状态总体上有三种场景。

场景一

获得synchronized隐式锁的线程，调用无参的Object.wait()方法。此时的线程会从RUNNABLE状态转换成WAITING状态。

场景二

调用无参数的Thread.join()方法。其中join()方法是一种线程的同步方法。例如，在threadA线程中调用threadB线程的join()方法，则threadA线程会等待threadB线程执行完。而threadA线程在等待threadB线程执行的过程中，其状态会从RUNNABLE转换到WAITING。当threadB执行完毕，threadA线程的状态则会从WAITING状态转换成RUNNABLE状态。

场景三

调用LockSupport.park()方法，当前线程会阻塞，线程的状态会从RUNNABLE转换成WAITING。调用LockSupport.unpark(Thread thread)可唤醒目标线程，目标线程的状态又会从WAITING状态转换到RUNNABLE。

RUNNABLE与TIMED_WAITING状态转换

总体上可以分为五种场景。

场景一

调用带超时参数的Thread.sleep(long millis)方法；

场景二

获得synchronized隐式锁的线程，调用带超时参数的Object.wait(long timeout)参数；

场景三

调用带超时参数的Thread.join(long millis)方法；

场景四

调用带超时参数的LockSupport.parkNanos(Object blocker, long deadline)方法；

场景五

调用带超时参数的LockSuppor.parkUntil(long deadline)方法。

从NEW到RUNNABLE状态

Java刚创建出来的Thread对象就是NEW状态，创建Thread对象主要有两种方法，一种是继承Thread对象，重写run()方法；另一种是实现Runnable接口，重写run()方法。

注意：这里说的是创建Thread对象的方法，而不是创建线程的方法，创建线程的方法包含创建Thread对象的方法。

继承Thread对象

```
public class ChildThread extends Thread{
    @Override
    public void run(){
        //线程中需要执行的逻辑
    }
}
//创建线程对象
ChildThread childThread = new ChildThread();
```

实现Runnable接口

```
public class ChildRunnable implements Runnable{
    @Override
    public void run(){
        //线程中需要执行的逻辑
    }
}
//创建线程对象
Thread childThread = new Thread(new ChildRunnable());
```

处于NEW状态的线程不会被操作系统调度，因此也就不会执行。Java中的线程要执行，就需要转换到RUNNABLE状态。从NEW状态转换到RUNNABLE状态，只需要调用线程对象的start()方法即可。

```
//创建线程对象
Thread childThread = new Thread(new ChildRunnable());
//调用start()方法使线程从NEW状态转换到RUNNABLE状态
childThread.start();
```

RUNNABLE到TERMINATED状态

线程执行完run()方法后，或者执行run()方法的时候抛出异常，都会终止，此时为TERMINATED状态。如果我们需要中断run()方法，可以调用interrupt()方法。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

实战案例篇

如何实现亿级流量下的分布式限流？这些理论你必须掌握！！

写在前面

> 在互联网应用中，高并发系统会面临一个重大的挑战，那就是大量流高并发访问，比如：天猫的双十一、京东618、秒杀、抢购促销等，这些都是典型的大流量高并发场景。关于秒杀，小伙伴们可以参见我的另一篇文章《[【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！](#)》
>
> 关注【冰河技术】微信公众号，解锁更多【高并发】专题文章。
>
> 注意：由于原文篇幅比较长，所以被拆分为：**理论、算法、实战（HTTP接口实战+分布式限流实战）**三大部分。

高并发系统限流

短时间内巨大的访问流量，我们如何让系统在处理高并发的同时还能保证自身系统的稳定性？有人会说，增加机器就可以了，因为我的系统是分布式的，所以可以只需要增加机器就可以解决问题了。但是，如果你通过增加机器还是不能解决这个问题怎么办呢？而且这种情况下又不能无限制的增加机器，服务器的硬件资源始终都是有限的，在有限的资源下，我们要应对这种大流量高并发的访问，就不得不采取一些其他的措施来保护我们的后端服务系统了，比如：缓存、异步、降级、限流、静态化等。

这里，我们先说说如何实现限流。

什么是限流？

在高并发系统中，限流通常指的是：对高并发访问或者请求进行限速或者对一个时间内的请求进行限速来保护我们的系统，一旦达到系统的限速规则（比如系统限制的请求速度），则可以采用下面的方式来处理这些请求。

- 拒绝服务（友好提示或者跳转到错误页面）。
- 排队或等待（比如秒杀系统）。
- 服务降级（返回默认的兜底数据）。

其实，就是对请求进行限速，比如10r/s，即每秒只允许10个请求，这样就限制了请求的速度。从某种意义上说，限流，其实就是在一定频率上进行量的限制。

限流一般用来控制系统服务请求的速率，比如：天猫双十一的限流，京东618的限流，12306的抢票等。

限流有哪些使用场景？

这里，我们来举一个例子，假设你做了一个商城系统，某个节假日的时候，突然发现提交订单的接口请求比平时请求量突然上涨了将近50倍，没多久提交订单的接口就超时并且抛出了异常，几乎不可用了。而且，因为订单接口超时不可用，还导致了系统其它服务出现故障。

我们该如何应对这种大流量场景呢？一种典型的处理方案就是限流。当然了，除了限流之外，还有其他的处理方案，我们这篇文章就主要讲限流。

- 对稀缺资源的秒杀、抢购；
- 对数据库的高并发读写操作，比如提交订单，瞬间往数据库插入大量的数据；

限流可以说是处理高并发问题的利器，有了限流就可以不用担心瞬间高峰流量压垮系统服务或者服务雪崩，最终做到有损服务而不是不服务。

使用限流同样需要注意的是：限流要评估好，测试好，否则会导致正常的访问被限流。

如何实现亿级流量下的分布式限流？这些算法你必须掌握！！

写在前面

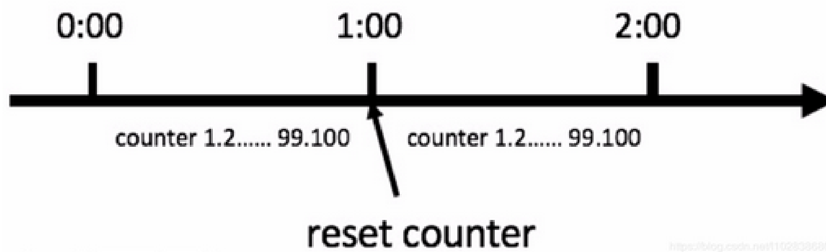
> 在互联网应用中，高并发系统会面临一个重大的挑战，那就是大量流高并发访问，比如：天猫的双十一、京东618、秒杀、抢购促销等，这些都是典型的大流量高并发场景。关于秒杀，小伙伴们可以参见我的另一篇文章《[【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！]》
>
> 关于【冰河技术】微信公众号，解锁更多【高并发】专题文章。
>
> 注意：由于原文篇幅比较长，所以被拆分为：理论、算法、实战（HTTP接口实战+分布式限流实战）三大部分。理论篇参见《[【高并发】如何实现亿级流量下的分布式限流？这些理论你必须掌握！！]》

计数器

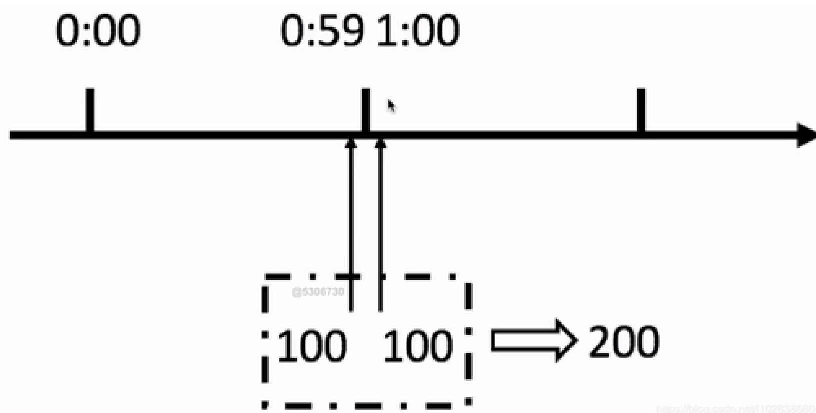
计数器法

限流算法中最简单粗暴的一种算法，例如，某一个接口1分钟内的请求不超过60次，我们可以在开始时设置一个计数器，每次请求时，这个计数器的值加1，如果这个这个计数器的值大于60并且与第一次请求的时间间隔在1分钟之内，那么说明请求过多；如果该请求与第一次请求的时间间隔大于1分钟，并且该计数器的值还在限流范围内，那么重置该计数器。

使用计数器还可以用来限制一定时间内的总并发数，比如数据库连接池、线程池、秒杀的并发数；计数器限流只要一定时间内的总请求数超过设定的阀值则进行限流，是一种简单粗暴的总数量限流，而不是平均速率限流。

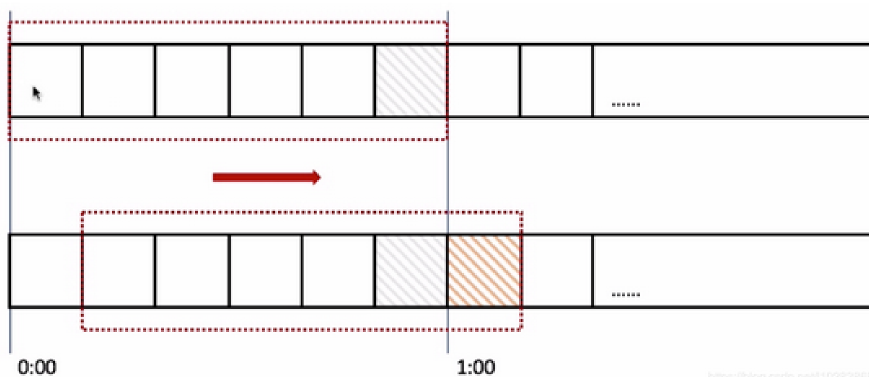


这个方法有一个致命问题：临界问题——当遇到恶意请求，在0:59时，瞬间请求100次，并且在1:00请求100次，那么这个用户在1秒内请求了200次，用户可以在重置节点突发请求，而瞬间超过我们设置的速率限制，用户可能通过算法漏洞击垮我们的应用。



这个问题我们可以使用滑动窗口解决。

滑动窗口

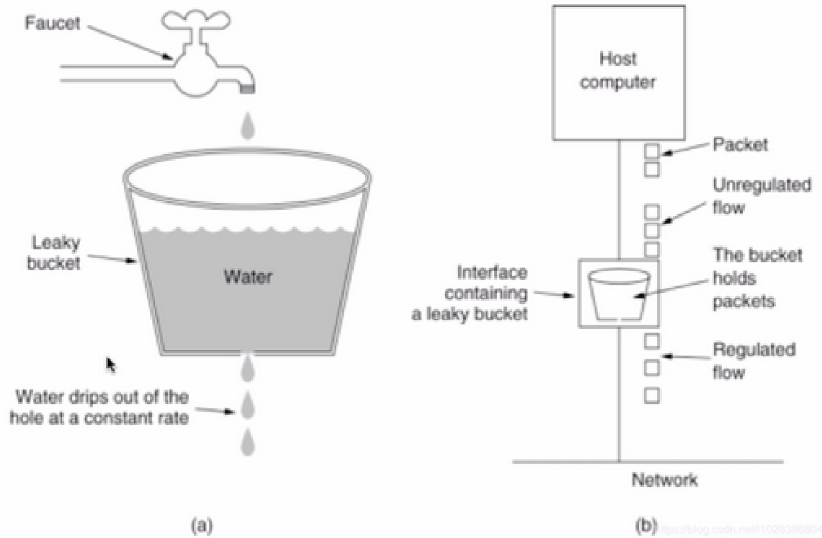


在上图中，整个红色矩形框是一个时间窗口，在我们的例子中，一个时间窗口就是1分钟，然后我们将时间窗口进行划分，如上图我们把滑动窗口划分为6格，所以每一格代表10秒，每超过10秒，我们的时间窗口就会向右滑动一格，每一格都有自己独立的计数器，例如：一个请求在0:35到达， 那么0:30到0:39的计数器会+1，那么滑动窗口是怎么解决临界点的问题呢？如上图，0:59到达的100个请求会在灰色区域格子中，而1：00到达的请求会在红色格子中，窗口会向右滑动一格，那么此时间窗口内的总请求数共200个，超过了限定的100，所以此时能够检测出来触发了限流。回头看看计数器算法，会发现，其实计数器算法就是窗口滑动算法，只不过计数器算法没有对时间窗口进行划分，所以是一格。

由此可见，当滑动窗口的格子划分越多，限流的统计就会越精确。

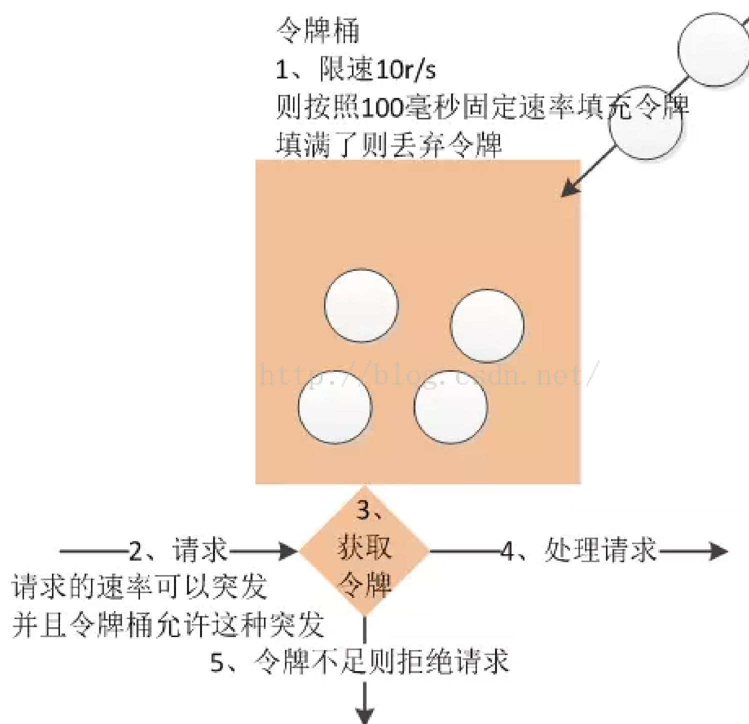
漏桶算法

算法的思路就是水（请求）先进入到漏桶里面，漏桶以恒定的速度流出，当水流的速度过大就会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。如下图所示。



漏桶算法不支持突发流量。

令牌桶算法



从上图中可以看出，令牌算法有点复杂，桶里存放着令牌token。桶一开始是空的，token以固定的速率r往桶里面填充，直到达到桶的容量，多余的token会被丢弃。每当一个请求过来时，就会尝试着移除一个token，如果没有token，请求无法通过。

令牌桶算法支持突发流量。

令牌桶算法实现

Guava框架提供了令牌桶算法的实现，可直接使用这个框架的RateLimiter类创建一个令牌桶限流器，比如：每秒放置的令牌桶的数量为5，那么RateLimiter对象可以保证1秒内不会放入超过5个令牌，并且以固定速率进行放置令牌，达到平滑输出的效果。

平滑流量示例

这里，我写了一个使用Guava框架实现令牌桶算法的示例，如下所示。

```
package io.binghe.limit.guava;

import com.google.common.util.concurrent.RateLimiter;

/**
 * @author binghe
```

```
 * @version 1.0.0
 * @description 令牌桶算法
 */
public class TokenBucketLimiter {
    public static void main(String[] args){
        //每秒钟生成5个令牌
        RateLimiter limiter = RateLimiter.create(5);

        //返回值表示从令牌桶中获取一个令牌所花费的时间，单位是秒
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
    }
}
```

代码的实现非常简单，就是使用Guava框架的RateLimiter类生成了一个每秒向桶中放入5个令牌的对象，然后不断从桶中获取令牌。我们先来运行下这段代码，输出的结果信息如下所示。

```
0.0
0.197294
0.191278
0.19997
0.199305
0.200472
0.200184
0.199417
0.200111
0.199759
```

从输出结果可以看出：第一次从桶中获取令牌时，返回的时间为0.0，也就是没耗费时间。之后每次从桶中获取令牌时，都会耗费一定的时间，这是为什么呢？按理说，向桶中放入了5个令牌后，再从桶中获取令牌也应该和第一次一样并不会花费时间啊！

因为在Guava的实现是这样的：我们使用`RateLimiter.create(5)`创建令牌桶对象时，表示每秒新增5个令牌，1秒等于1000毫秒，也就是每隔200毫秒向桶中放入一个令牌。

当我们运行程序时，程序运行到`RateLimiter limiter = RateLimiter.create(5);`时，就会向桶中放入一个令牌，当程序运行到第一个`System.out.println(limiter.acquire(1));`时，由于桶中已经存在一个令牌，直接获取这个令牌，并没有花费时间。然而程序继续向下执行时，由于程序会每隔200毫秒向桶中放入一个令牌，所以，获取令牌时，花费的时间几乎都是200毫秒左右。

突发流量示例

我们再来看一个突发流量的示例，代码示例如下所示。

```
package io.binghe.limit.guava;

import com.google.common.util.concurrent.RateLimiter;

/**
 * @author binghe
 * @version 1.0.0
 * @description 令牌桶算法
 */
public class TokenBucketLimiter {
    public static void main(String[] args){
        //每秒钟生成5个令牌
        RateLimiter limiter = RateLimiter.create(5);

        //返回值表示从令牌桶中获取一个令牌所花费的时间，单位是秒
        System.out.println(limiter.acquire(50));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
```

```
    }
}
```

上述代码表示的含义为：每秒向桶中放入5个令牌，第一次从桶中获取50个令牌，也就是我们说的突发流量，后续每次从桶中获取5个令牌。接下来，我们运行上述代码看下效果。

```
0.0
9.998409
0.99109
1.000148
0.999752
```

运行代码时，会发现当命令行打印出0.0后，会等很久才会打印出后面的输出结果。

程序每秒钟向桶中放入5个令牌，当程序运行到 `RateLimiter limiter = RateLimiter.create(5);` 时，就会向桶中放入令牌。当运行到 `System.out.println(limiter.acquire(50));` 时，发现很快就会获取到令牌，花费了0.0秒。接下来，运行到第一个 `System.out.println(limiter.acquire(5));` 时，花费了9.998409秒。小伙伴们可以思考下，为什么这里会花费10秒中的时间呢？

这是因为我们使用 `RateLimiter limiter = RateLimiter.create(5);` 代码向桶中放入令牌时，一秒钟放入5个，而 `System.out.println(limiter.acquire(50));` 需要获取50个令牌，也就是获取50个令牌需要花费10秒钟时间，这是因为程序向桶中放入50个令牌需要10秒钟。程序第一次从桶中获取令牌时，很快就获取到了。而第二次获取令牌时，花费了将近10秒的时间。

Guava框架支持突发流量，但是在突发流量之后再次请求时，会被限速，也就是说：在突发流量之后，再次请求时，会弥补处理突发请求所花费的时间。所以，我们的突发示例程序中，在一次从桶中获取50个令牌后，再次从桶中获取令牌，则会花费10秒左右的时间。

Guava令牌桶算法的特点

- RateLimiter使用令牌桶算法，会进行令牌的累积，如果获取令牌的频率比较低，则不会导致等待，直接获取令牌。
- RateLimiter由于会累积令牌，所以可以应对突发流量。也就是说如果同时请求5个令牌，由于此时令牌桶中有累积的令牌，能够快速响应请求。
- RateLimiter在没有足够的令牌发放时，采用的是滞后的方式进行处理，也就是前一个请求获取令牌所需要等待的时间由下一次请求来承受和弥补，也就是代替前一个请求进行等待。（这里，小伙伴们要好好理解下）

亿级流量场景下如何为HTTP接口限流？看完我懂了！！

写在前面

> 在互联网应用中，高并发系统会面临一个重大的挑战，那就是大量流高并发访问，比如：天猫的双十一、京东618、秒杀、抢购促销等，这些都是典型的大流量高并发场景。关于秒杀，小伙伴们可以参见我的另一篇文章《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》
>
> 关于【冰河技术】微信公众号，解锁更多【高并发】专题文章。
>
> 注意：由于原文篇幅比较长，所以被拆分为：理论、算法、实战（HTTP接口实战+分布式限流实战）三大部分。
>
> 理论篇：《【高并发】如何实现亿级流量下的分布式限流？这些理论你必须掌握！！》
>
> 算法篇：《【高并发】如何实现亿级流量下的分布式限流？这些算法你必须掌握！！》
>
> 项目源码已提交到github：https://github.com/sunshinelyz/mykit-ratelimiter

HTTP接口限流实战

这里，我们实现Web接口限流，具体方式为：使用自定义注解封装基于令牌桶限流算法实现接口限流。

不使用注解实现接口限流

搭建项目

这里，我们使用SpringBoot项目来搭建Http接口限流项目，SpringBoot项目本质上还是一个Maven项目。所以，小伙伴们可以直接创建一个Maven项目，我这里的项目名称为mykit-ratelimiter-test。接下来，在pom.xml文件中添加如下依赖使项目构建为一个SpringBoot项目。

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
    </parent>
```

```xml
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.mykit.limiter</groupId>
    <artifactId>mykit-ratelimiter-test</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>mykit-ratelimiter-test</name>

    <properties>
        <guava.version>28.2-jre</guava.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-undertow</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>

        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>${guava.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.1</version><!--$NO-MVN-MAN-VER$-->
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

可以看到，我在项目中除了引用了SpringBoot相关的Jar包外，还引用了guava框架，版本为28.2-jre。

创建核心类

这里，我主要是模拟一个支付接口的限流场景。首先，我们定义一个PayService接口和MessageService接口。PayService接口主要用于模拟后续的支付业务，MessageService接口模拟发送消息。接口的定义分别如下所示。

- PayService

```
package io.mykit.limiter.service;
import java.math.BigDecimal;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟支付
 */
public interface PayService {
    int pay(BigDecimal price);
}
```

- MessageService

```
package io.mykit.limiter.service;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟发送消息服务
 */
public interface MessageService {
    boolean sendMessage(String message);
}
```

接下来，创建二者的实现类，分别如下。

- MessageServiceImpl

```
package io.mykit.limiter.service.impl;
import io.mykit.limiter.service.MessageService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟实现发送消息
 */
@Service
public class MessageServiceImpl implements MessageService {
    private final Logger logger = LoggerFactory.getLogger(MessageServiceImpl.class);
    @Override
    public boolean sendMessage(String message) {
        logger.info("发送消息成功===>>" + message);
        return true;
    }
}
```

- PayServiceImpl

```
package io.mykit.limiter.service.impl;
import io.mykit.limiter.service.PayService;
```

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import java.math.BigDecimal;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟支付
 */
@Service
public class PayServiceImpl implements PayService {
    private final Logger logger = LoggerFactory.getLogger(PayServiceImpl.class);
    @Override
    public int pay(BigDecimal price) {
        logger.info("支付成功===>>" + price);
        return 1;
    }
}
```

由于是模拟支付和发送消息，所以，我在具体实现的方法中打印出了相关的日志，并没有实现具体的业务逻辑。

接下来，就是创建我们的Controller类PayController，在PayController类的接口pay()方法中使用了限流，每秒钟向桶中放入2个令牌，并且客户端从桶中获取令牌，如果在500毫秒内没有获取到令牌的话，我们可以则直接走服务降级处理。

PayController的代码如下所示。

```java
package io.mykit.limiter.controller;
import com.google.common.util.concurrent.RateLimiter;
import io.mykit.limiter.service.MessageService;
import io.mykit.limiter.service.PayService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.math.BigDecimal;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试接口限流
 */
@RestController
public class PayController {
    private final Logger logger = LoggerFactory.getLogger(PayController.class);
    /**
     * RateLimiter的create()方法中传入一个参数，表示以固定的速率2r/s，即以每秒2个令牌的速率向桶中放入令牌
     */
    private RateLimiter rateLimiter = RateLimiter.create(2);

    @Autowired
    private MessageService messageService;
    @Autowired
    private PayService payService;
    @RequestMapping("/boot/pay")
    public String pay(){
        //记录返回接口
        String result = "";
        //限流处理，客户端请求从桶中获取令牌，如果在500毫秒没有获取到令牌，则直接走服务降级处理
        boolean tryAcquire = rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS);
        if (!tryAcquire){
            result = "请求过多，降级处理";
            logger.info(result);
            return result;
        }
        int ret = payService.pay(BigDecimal.valueOf(100.0));
        if(ret > 0){
            result = "支付成功";
            return result;
        }
```

```
            result = "支付失败，再试一次吧...";
            return result;
        }
    }
```

最后，我们来创建mykit-ratelimiter-test项目的核心启动类，如下所示。

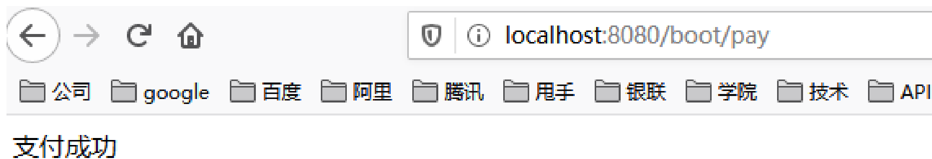
```
package io.mykit.limiter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author binghe
 * @version 1.0.0
 * @description 项目启动类
 */
@SpringBootApplication
public class MykitLimiterApplication {

    public static void main(String[] args){
        SpringApplication.run(MykitLimiterApplication.class, args);
    }
}
```

至此，我们不使用注解方式实现限流的Web应用就基本完成了。

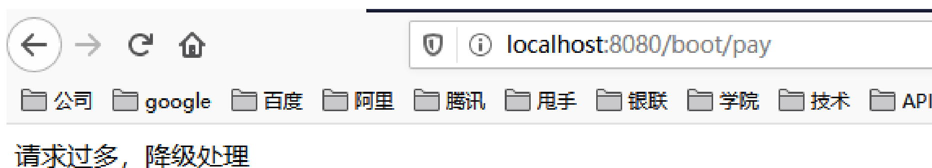
运行项目

项目创建完成后，我们来运行项目，运行SpringBoot项目比较简单，直接运行MykitLimiterApplication类的main()方法即可。

项目运行成功后，我们在浏览器地址栏输入链接：http://localhost:8080/boot/pay。页面会输出"支付成功"的字样，说明项目搭建成功了。如下所示。



此时，我只访问了一次，并没有触发限流。接下来，我们不停的刷浏览器，此时，浏览器会输出"支付失败，再试一次吧..."的字样，如下所示。



在PayController类中还有一个sendMessage()方法，模拟的是发送消息的接口，同样使用了限流操作，具体代码如下所示。

```
@RequestMapping("/boot/send/message")
public String sendMessage(){
    //记录返回接口
    String result = "";
    //限流处理，客户端请求从桶中获取令牌，如果在500毫秒没有获取到令牌，则直接走服务降级处理
    boolean tryAcquire = rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS);
    if (!tryAcquire){
        result = "请求过多，降级处理";
        logger.info(result);
        return result;
    }
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "消息发送成功";
        return result;
    }
```

```
        result = "消息发送失败，再试一次吧...";
        return result;
    }
```

sendMessage()方法的代码逻辑和运行效果与pay()方法相同，我就不再浏览器访问 http://localhost:8080/boot/send/message 地址的访问效果了，小伙伴们可以自行验证。

不使用注解实现限流缺点

通过对项目的编写，我们可以发现，当在项目中对接口进行限流时，不使用注解进行开发，会导致代码出现大量冗余，每个方法中几乎都要写一段相同的限流逻辑，代码十分冗余。

如何解决代码冗余的问题呢？我们可以使用自定义注解进行实现。

使用注解实现接口限流

使用自定义注解，我们可以将一些通用的业务逻辑封装到注解的切面中，在需要添加注解业务逻辑的方法上加上相应的注解即可。针对我们这个限流的实例来说，可以基于自定义注解实现。

实现自定义注解

实现，我们来创建一个自定义注解，如下所示。

```
package io.mykit.limiter.annotation;
import java.lang.annotation.*;
/**
 * @author binghe
 * @version 1.0.0
 * @description 实现限流的自定义注解
 */
@Target(value = ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRateLimiter {
    //向令牌桶放入令牌的速率
    double rate();
    //从令牌桶获取令牌的超时时间
    long timeout() default 0;
}
```

自定义注解切面实现

接下来，我们还要实现一个切面类MyRateLimiterAspect，如下所示。

```
package io.mykit.limiter.aspect;

import com.google.common.util.concurrent.RateLimiter;
import io.mykit.limiter.annotation.MyRateLimiter;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 一般限流切面类
 */
@Aspect
@Component
public class MyRateLimiterAspect {
```

```java
    private RateLimiter rateLimiter = RateLimiter.create(2);

    @Pointcut("execution(public * io.mykit.limiter.controller.*.*(..))")
    public void pointcut(){

    }

    /**
     * 核心切面方法
     */
    @Around("pointcut()")
    public Object process(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
        MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();

        //使用反射获取方法上是否存在@MyRateLimiter注解
        MyRateLimiter myRateLimiter = signature.getMethod().getDeclaredAnnotation(MyRateLimiter.class);
        if(myRateLimiter == null){
            //程序正常执行，执行目标方法
            return proceedingJoinPoint.proceed();
        }
        //获取注解上的参数
        //获取配置的速率
        double rate = myRateLimiter.rate();
        //获取客户端等待令牌的时间
        long timeout = myRateLimiter.timeout();

        //设置限流速率
        rateLimiter.setRate(rate);

        //判断客户端获取令牌是否超时
        boolean tryAcquire = rateLimiter.tryAcquire(timeout, TimeUnit.MILLISECONDS);
        if(!tryAcquire){
            //服务降级
            fullback();
            return null;
        }
        //获取到令牌，直接执行
        return proceedingJoinPoint.proceed();

    }

    /**
     * 降级处理
     */
    private void fullback() {
        response.setHeader("Content-type", "text/html;charset=UTF-8");
        PrintWriter writer = null;
        try {
            writer =  response.getWriter();
            writer.println("出错了，重试一次试试？");
            writer.flush();;
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(writer != null){
                writer.close();
            }
        }
    }
}
```

自定义切面的功能比较简单，我就不细说了，大家有啥问题可以关注【冰河技术】微信公众号来进行提问。

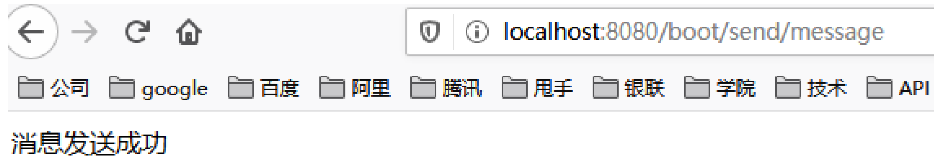
接下来，我们改造下PayController类中的sendMessage()方法，修改后的方法片段代码如下所示。

```java
@MyRateLimiter(rate = 1.0, timeout = 500)
@RequestMapping("/boot/send/message")
public String sendMessage(){
    //记录返回接口
    String result = "";
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "消息发送成功";
        return result;
    }
    result = "消息发送失败，再试一次吧...";
    return result;
}
```

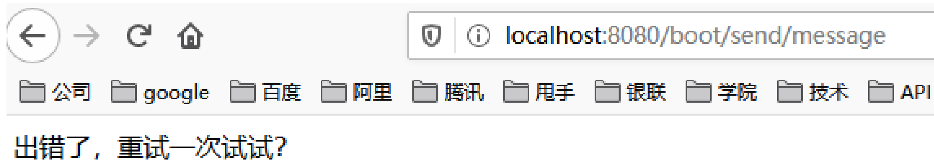
运行部署项目

部署项目比较简单，只需要运行MykitLimiterApplication类下的main()方法即可。这里，为了简单，我们还是从浏览器中直接输入链接地址来进行访问

效果如下所示。



接下来，我们不断的刷新浏览器。会出现"消息发送失败，再试一次吧.."的字样，说明已经触发限流操作。



基于限流算法实现限流的缺点

上面介绍的限流方式都只能用于单机部署的环境中，如果将应用部署到多台服务器进行分布式、集群，则上面限流的方式就不适用了，此时，我们需要使用分布式限流。至于在分布式场景下，如何实现限流操作，我们就在下一篇中进行介绍。

亿级流量场景下如何实现分布式限流？看完我彻底懂了！！

写在前面

> 在互联网应用中，高并发系统会面临一个重大的挑战，那就是大量流高并发访问，比如：天猫的双十一、京东618、秒杀、抢购促销等，这些都是典型的大流量高并发场景。关于秒杀，小伙伴们可以参见我的另一篇文章《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》
>
> 关于【冰河技术】微信公众号，解锁更多【高并发】专题文章。
>
> 注意：由于原文篇幅比较长，所以被拆分为：理论、算法、实战（HTTP接口实战+分布式限流实战）三大部分。
>
> 理论篇：《【高并发】如何实现亿级流量下的分布式限流？这些理论你必须掌握！！》
>
> 算法篇：《【高并发】如何实现亿级流量下的分布式限流？这些算法你必须掌握！！》
>
> 项目源码已提交到github：https://github.com/sunshinelyz/mykit-ratelimiter

本文是在《【高并发】亿级流量场景下如何为HTTP接口限流？看完我懂了！！》一文的基础上进行实现，有关项目的搭建可参见《【高并发】亿级流量场景下如何为HTTP接口限流？看完我懂了！！》一文的内容。小伙伴们可以关注【冰河技术】微信公众号来阅读上述文章。

前面介绍的限流方案有一个缺陷就是：它不是全局的，不是分布式的，无法很好的应对分布式场景下的大流量冲击。那么，接下来，我们就介绍下如何实现亿级流量下的分布式限流。

分布式限流的关键就是需要将限流服务做成全局的，统一的。可以采用Redis+Lua技术实现，通过这种技术可以实现高并发和高性能的限流。

Lua是一种轻量小巧的脚本编程语言，用标准的C语言编写的开源脚本，其设计的目的是为了嵌入到应用程序中，为应用程序提供灵活的扩展和定制功能。

Redis+Lua脚本实现分布式限流思路

我们可以使用Redia+Lua脚本的方式来对我们的分布式系统进行统一的全局限流，Redis+Lua实现的Lua脚本：

```
local key = KEYS[1]  --限流KEY(一秒一个)
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
    return 0
else --请求数+1，并设置2秒过期
    redis.call("INCRBY", key, "1")
    redis.call("expire", key "2")
    return 1
end
```

我们可以按照如下的思路来理解上述Lua脚本代码。

（1）在Lua脚本中，有两个全局变量，用来接收Redis应用端传递的键和其他参数，分别为：KEYS、ARGV；

（2）在应用端传递KEYS时是一个数组列表，在Lua脚本中通过索引下标方式获取数组内的值。

（3）在应用端传递ARGV时参数比较灵活，可以是一个或多个独立的参数，但对应到Lua脚本中统一用ARGV这个数组接收，获取方式也是通过数组下标获取。

（4）以上操作是在一个Lua脚本中，又因为我当前使用的是Redis 5.0版本（Redis 6.0支持多线程），执行的请求是单线程的，因此，Redis+Lua的处理方式是线程安全的，并且具有原子性。

这里，需要注意一个知识点，那就是原子性操作：如果一个操作时不可分割的，是多线程安全的，我们就称为原子性操作。

接下来，我们可以使用如下Java代码来判断是否需要限流。

```
//List设置Lua的KEYS[1]
String key = "ip:" + System.currentTimeMillis() / 1000;
List<String> keyList = Lists.newArrayList(key);

//List设置Lua的ARGV[1]
List<String> argvList = Lists.newArrayList(String.valueOf(value));

//调用Lua脚本并执行
List result = stringRedisTemplate.execute(redisScript, keyList, argvList)
```

至此，我们简单的介绍了使用Redis+Lua脚本实现分布式限流的总体思路，并给出了Lua脚本的核心代码和Java程序调用Lua脚本的核心代码。接下来，我们就动手写一个使用Redis+Lua脚本实现的分布式限流案例。

Redis+Lua脚本实现分布式限流案例

这里，我们和在《[【高并发】亿级流量场景下如何为HTTP接口限流？看完我懂了！！]》一文中的实现方式类似，也是通过自定义注解的形式来实现分布式、大流量场景下的限流，只不过这里我们使用了Redis+Lua脚本的方式实现了全局统一的限流模式。接下来，我们就一起手动实现这个案例。

创建注解

首先，我们在项目中，定义个名称为MyRedisLimiter的注解，具体代码如下所示。

```
package io.mykit.limiter.annotation;
import org.springframework.core.annotation.AliasFor;
import java.lang.annotation.*;
/**
 * @author binghe
 * @version 1.0.0
 * @description 自定义注解实现分布式限流
 */
@Target(value = ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRedisLimiter {
    @AliasFor("limit")
    double value() default Double.MAX_VALUE;
    double limit() default Double.MAX_VALUE;
}
```

在MyRedisLimiter注解内部，我们为value属性添加了别名limit，在我们真正使用@MyRedisLimiter注解时，即可以使用@MyRedisLimiter(10)，也可以使用@MyRedisLimiter(value=10)，还可以使用@MyRedisLimiter(limit=10)。

创建切面类

创建注解后，我们就来创建一个切面类MyRedisLimiterAspect，MyRedisLimiterAspect类的作用主要是解析@MyRedisLimiter注解，并且执行限流的规则。这样，就不需要我们在每个需要限流的方法中执行具体的限流逻辑了，只需要我们在需要限流的方法上添加@MyRedisLimiter注解即可，具体代码如下所示。

```
package io.mykit.limiter.aspect;
import com.google.common.collect.Lists;
import io.mykit.limiter.annotation.MyRedisLimiter;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.scripting.support.ResourceScriptSource;
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.util.List;

/**
 * @author binghe
 * @version 1.0.0
 * @description MyRedisLimiter注解的切面类
 */
@Aspect
@Component
public class MyRedisLimiterAspect {
    private final Logger logger = LoggerFactory.getLogger(MyRedisLimiter.class);
    @Autowired
    private HttpServletResponse response;
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private DefaultRedisScript<List> redisScript;

    @PostConstruct
    public void init(){
        redisScript = new DefaultRedisScript<List>();
        redisScript.setResultType(List.class);
        redisScript.setScriptSource(new ResourceScriptSource(new ClassPathResource(("limit.lua"))));
    }

    @Pointcut("execution(public * io.mykit.limiter.controller.*.*(..))")
    public void pointcut(){

    }

    @Around("pointcut()")
    public Object process(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
        MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();
        //使用反射获取MyRedisLimiter注解
        MyRedisLimiter myRedisLimiter =
signature.getMethod().getDeclaredAnnotation(MyRedisLimiter.class);
        if(myRedisLimiter == null){
            //正常执行方法
            return proceedingJoinPoint.proceed();
        }
        //获取注解上的参数，获取配置的速率
        double value = myRedisLimiter.value();
```

```java
        //List设置Lua的KEYS[1]
        String key = "ip:" + System.currentTimeMillis() / 1000;
        List<String> keyList = Lists.newArrayList(key);

        //List设置Lua的ARGV[1]
        List<String> argvList = Lists.newArrayList(String.valueOf(value));

        //调用Lua脚本并执行
        List result = stringRedisTemplate.execute(redisScript, keyList, String.valueOf(value));
        logger.info("Lua脚本的执行结果：" + result);

        //Lua脚本返回0，表示超出流量大小，返回1表示没有超出流量大小。
        if("0".equals(result.get(0).toString())){
            fullBack();
            return null;
        }

        //获取到令牌，继续向下执行
        return proceedingJoinPoint.proceed();
    }

    private void fullBack() {
        response.setHeader("Content-Type" ,"text/html;charset=UTF8");
        PrintWriter writer = null;
        try{
            writer = response.getWriter();
            writer.println("回退失败，请稍后阅读。。。");
            writer.flush();
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            if(writer != null){
                writer.close();
            }
        }
    }
}
```

上述代码会读取项目classpath目录下的limit.lua脚本文件来确定是否执行限流的操作，调用limit.lua文件执行的结果返回0则表示执行限流逻辑，否则不执行限流逻辑。既然，项目中需要使用Lua脚本，那么，接下来，我们就需要在项目中创建Lua脚本。

创建limit.lua脚本文件

在项目的classpath目录下创建limit.lua脚本文件，文件的内容如下所示。

```lua
local key = KEYS[1]  --限流KEY(一秒一个)
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
    return 0
else --请求数+1，并设置2秒过期
    redis.call("INCRBY", key, "1")
    redis.call("expire", key "2")
    return 1
end
```

limit.lua脚本文件的内容比较简单，这里就不再赘述了。

接口添加注解

注解类、解析注解的切面类、Lua脚本文件都已经准备好。那么，接下来，我们在PayController类中在sendMessage2()方法上添加@MyRedisLimiter注解，并且将limit属性设置为10，如下所示。

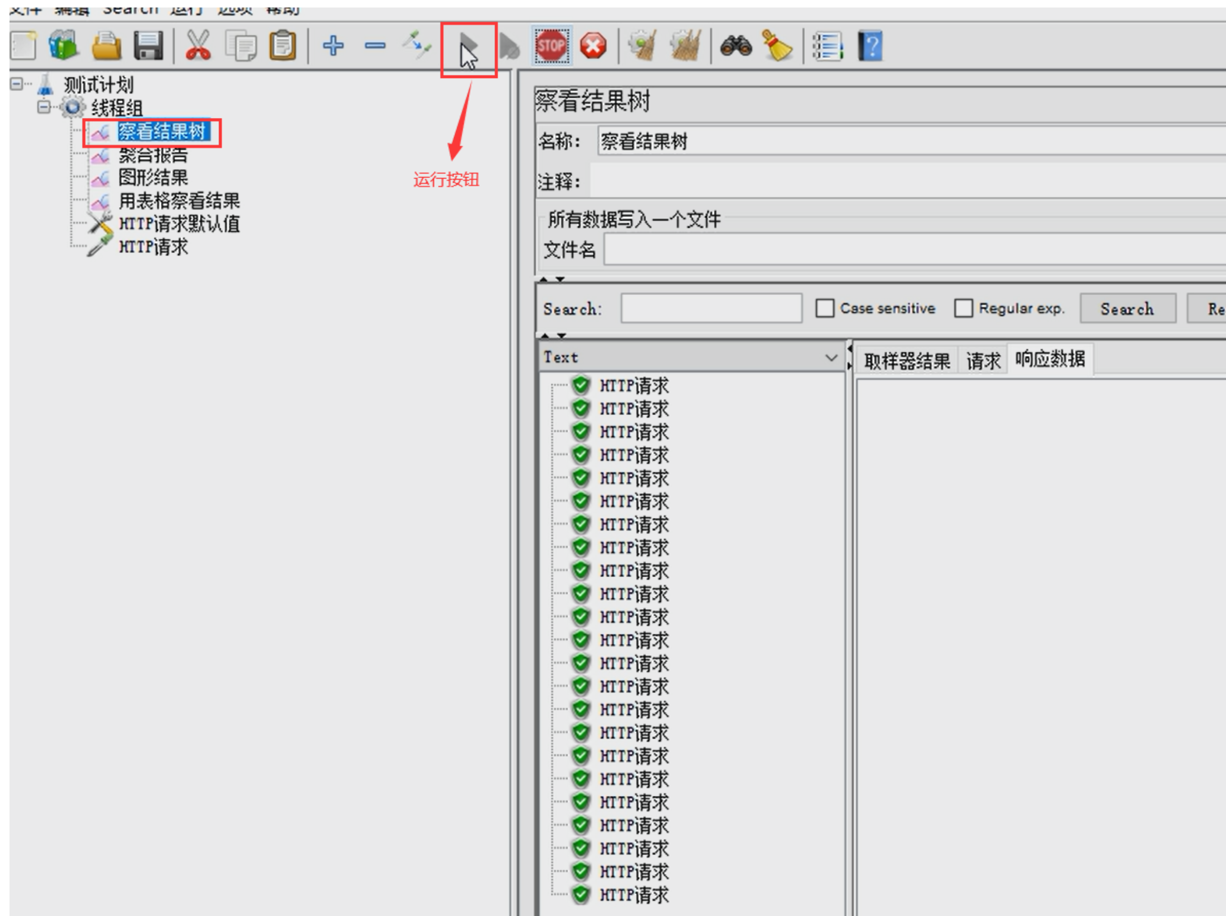
```java
@MyRedisLimiter(limit = 10)
@RequestMapping("/boot/send/message2")
public String sendMessage2(){
    //记录返回接口
    String result = "";
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "短信发送成功！";
        return result;
    }
    result = "哎呀，服务器开小差了，请再试一下吧";
    return result;
}
```

此处，我们限制了sendMessage2()方法，每秒钟最多只能处理10个请求。那么。接下来，我们就使用JMeter对sendMessage2()进行测试。

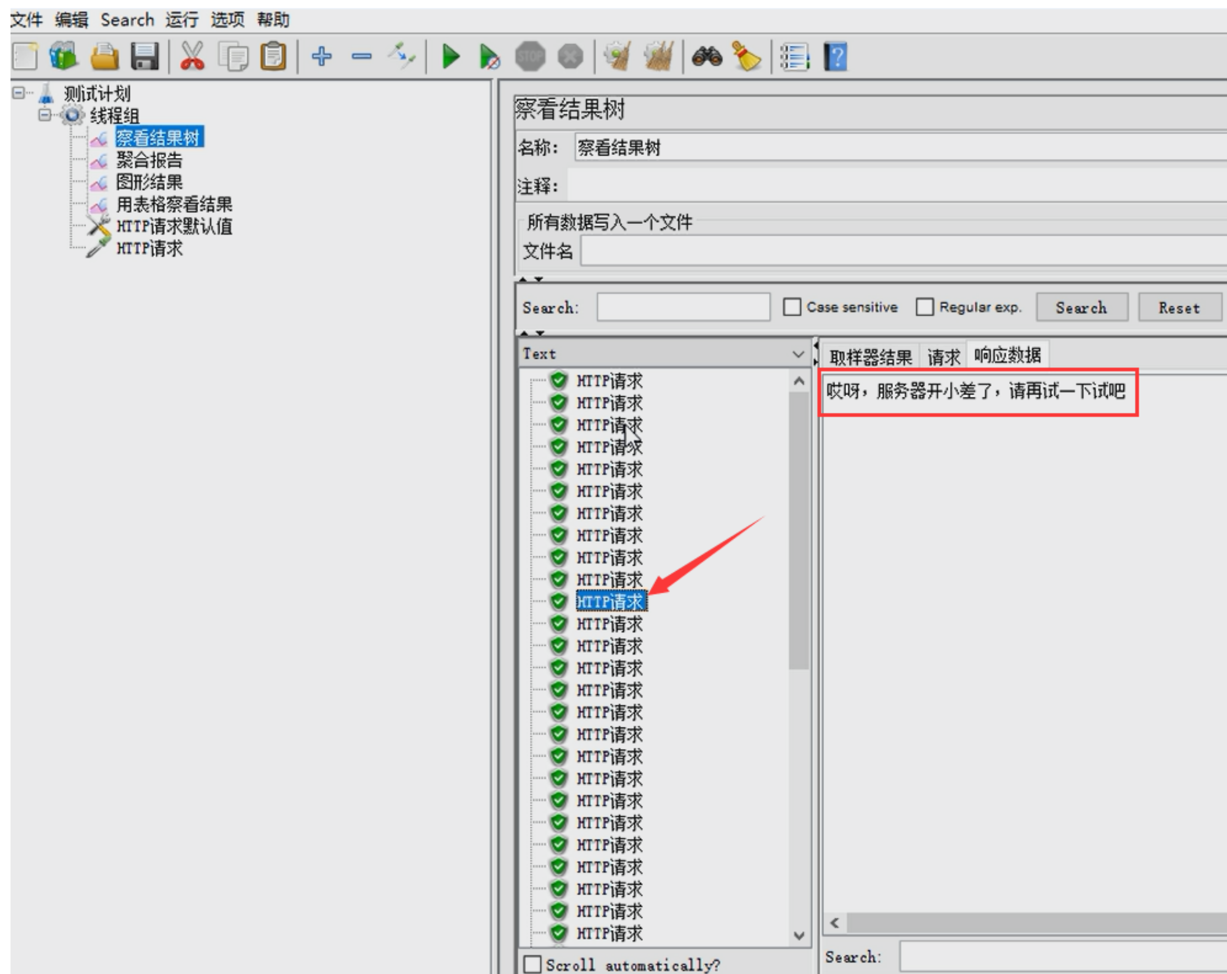
测试分布式限流

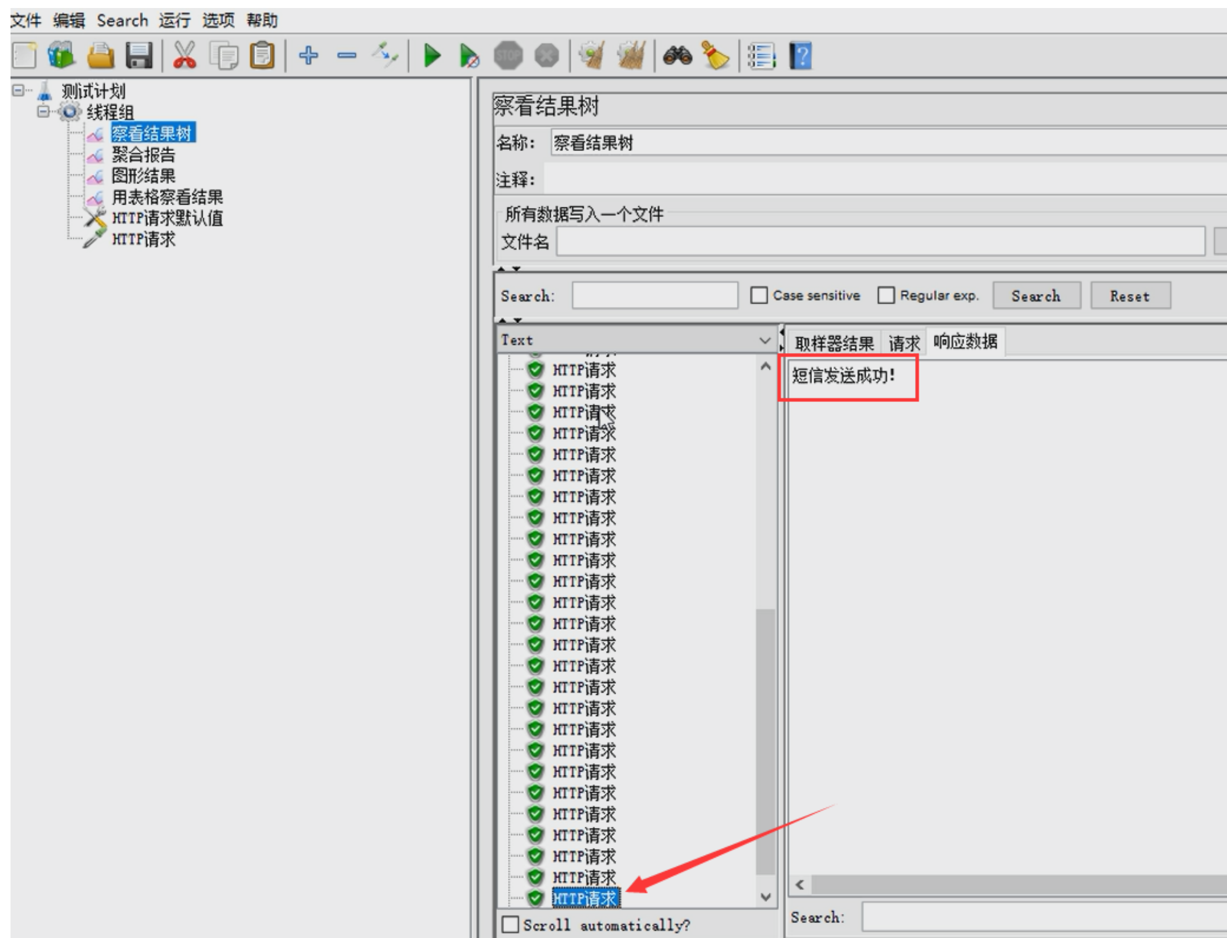
此时，我们使用JMeter进行压测，这里，我们配置的线程数为50，也就是说：会有50个线程同时访问我们写的接口。JMeter的配置如下所示。

保存并运行Jemeter，如下所示。



运行完成后，我们来查看下JMeter的测试结果，如下所示。





从测试结果可以看出，测试中途有部分接口的访问返回了“哎呀，服务器开小差了，请再试一下吧”，说明接口被限流了。而再往后，又有部分接口成功返回了“短信发送成功！”的字样。这是因为我们设置的是接口每秒最多接受10次请求，在第一秒内访问接口时，前面的10次请求成功返回“短信发送成功！”的字样，后面再访问接口就会返回“哎呀，服务器开小差了，请再试一下吧”。而后面的请求又返回了“短信发送成功！”的字样，说明后面的请求已经是在第二秒的时候调用的接口。

我们使用Redis+Lua脚本的方式实现的限流方式，可以将Java程序进行集群部署，这种方式实现的是全局的统一的限流，无论客户端访问的是集群中的哪个节点，都会对访问进行计数并实现最终的限流效果。

这种思想就有点像分布式锁了，小伙伴们可以关注【冰河技术】微信公众号阅读我写的一篇《【高并发】高并发分布式锁架构解密，不是所有的锁都是分布式锁！！》来深入理解如何实现真正线程安全的分布式锁，此文章，以循序渐进的方式深入剖析了实现分布式锁过程中的各种坑和解决方案，让你真正理解什么才是分布式锁。

Nginx+Lua实现分布式限流

Nginx+Lua实现分布式限流，通常会用在应用的入口处，也就是对系统的流量入口进行限流。这里，我们也以一个实际案例的形式来说明如何使用Nginx+Lua来实现分布式限流。

首先，我们需要创建一个Lua脚本，脚本文件的内容如下所示。

```
local locks = require "resty.lock"

local function acquire()
    local lock =locks:new("locks")
    local elapsed, err =lock:lock("limit_key") --互斥锁
    local limit_counter =ngx.shared.limit_counter --计数器

    local key = "ip:" ..os.time()
    local limit = 5 --限流大小
    local current =limit_counter:get(key)

    if current ~= nil and current + 1> limit then --如果超出限流大小
       lock:unlock()
       return 0
    end
    if current == nil then
       limit_counter:set(key, 1, 1) --第一次需要设置过期时间，设置key的值为1，过期时间为1秒
    else
        limit_counter:incr(key, 1) --第二次开始加1即可
    end
    lock:unlock()
    return 1
end
ngx.print(acquire())
```

实现中我们需要使用lua-resty-lock互斥锁模块来解决原子性问题(在实际工程中使用时请考虑获取锁的超时问题)，并使用ngx.shared.DICT共享字典来实现计数器。如果需要限流则返回0，否则返回1。使用时需要先定义两个共享字典（分别用来存放锁和计数器数据）。

接下来，需要在Nginx的nginx.conf配置文件中定义数据字典，如下所示。

```
http {
    ……
    lua_shared_dict locks 10m;
    lua_shared_dict limit_counter 10m;
}
```

灵魂拷问

说到这里，相信有很多小伙伴可能会问：如果应用并发量非常大，那么，Redis或者Nginx能不能扛的住呢？

可以这么说：Redis和Nginx基本都是高性能的互联网组件，对于一般互联网公司的高并发流量是完全没有问题的。为什么这么说呢？咱们继续往下看。

如果你的应用流量真的非常大，可以通过一致性哈希将分布式限流进行分片，还可以将限流降级为应用级限流；解决方案也非常多，可以根据实际情况进行调整，使用Redis+Lua的方式进行限流，是可以稳定达到对上亿级别的高并发流量进行限流的（笔者亲身经历）。

需要注意的是：面对高并发系统，尤其是这种流量上千万、上亿级别的高并发系统，我们不可能只用限流这一招，还要加上其他的一些措施，

对于分布式限流，目前遇到的场景是业务上的限流，而不是流量入口的限流。对于流量入口的限流，应该在接入层来完成。

对于秒杀场景来说，可以在流量入口处进行限流，小伙伴们可以关注【冰河技术】微信公众号，来阅读我写的《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》一文，来深入理解如何架构一个高并发秒杀系统

如何实现亿级流量下的分布式限流?

高并发系统限流

短时间内巨大的访问流量，我们如何让系统在处理高并发的同时还能保证自身系统的稳定性？有人会说，增加机器就可以了，因为我的系统是分布式的，所以可以只需要增加机器就可以解决问题了。但是，如果你通过增加机器还是不能解决这个问题怎么办呢？而且这种情况下又不能无限制的增加机器，服务器的硬件资源始终都是有限的，在有限的资源下，我们要应对这种大流量高并发的访问，就不得不采取一些其他的措施来保护我们的后端服务系统了，比如：缓存、异步、降级、限流、静态化等。

这里，我们先说说如何实现限流。

什么是限流?

在高并发系统中，限流通常指的是：对高并发访问或者请求进行限速或者对一个时间内的请求进行限速来保护我们的系统，一旦达到系统的限速规则（比如系统限制的请求速度），则可以采用下面的方式来处理这些请求。

- 拒绝服务（友好提示或者跳转到错误页面）。
- 排队或等待（比如秒杀系统）。
- 服务降级（返回默认的兜底数据）。

其实，就是对请求进行限速，比如10r/s，即每秒只允许10个请求，这样就限制了请求的速度。从某种意义上说，限流，其实就是在一定频率上进行量的限制。

限流一般用来控制系统服务请求的速率，比如：天猫双十一的限流，京东618的限流，12306的抢票等。

限流有哪些使用场景?

这里，我们来举一个例子，假设你做了一个商城系统，某个节假日的时候，突然发现提交订单的接口请求比平时请求量突然上涨了将近50倍，没多久提交订单的接口就超时并且抛出了异常，几乎不可用了。而且，因为订单接口超时不可用，还导致了系统其它服务出现故障。

我们该如何应对这种大流量场景呢？一种典型的处理方案就是限流。当然了，除了限流之外，还有其他的处理方案，我们这篇文章就主要讲限流。

- 对稀缺资源的秒杀、抢购；
- 对数据库的高并发读写操作，比如提交订单，瞬间往数据库插入大量的数据；

限流可以说是处理高并发问题的利器，有了限流就可以不用担心瞬间高峰流量压垮系统服务或者服务雪崩，最终做到有损服务而不是不服务。

使用限流同样需要注意的是：限流要评估好，测试好，否则会导致正常的访问被限流。

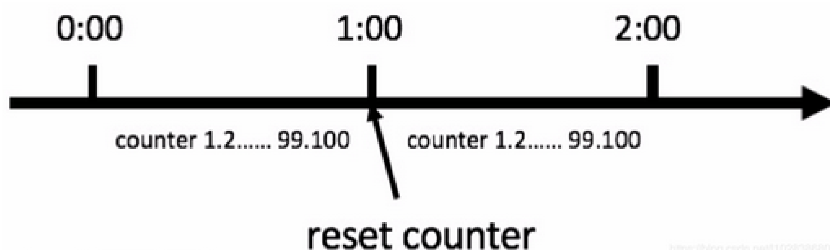
常见的限流算法

计数器

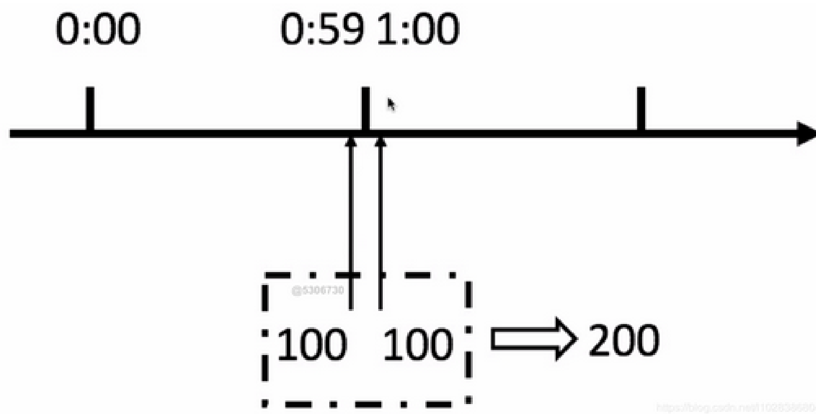
计数器法

限流算法中最简单粗暴的一种算法，例如，某一个接口1分钟内的请求不超过60次，我们可以在开始时设置一个计数器，每次请求时，这个计数器的值加1，如果这个这个计数器的值大于60并且与第一次请求的时间间隔在1分钟之内，那么说明请求过多；如果该请求与第一次请求的时间间隔大于1分钟，并且该计数器的值还在限流范围内，那么重置该计数器。

使用计数器还可以用来限制一定时间内的总并发数，比如数据库连接池、线程池、秒杀的并发数；计数器限流只要一定时间内的总请求数超过设定的阀值则进行限流，是一种简单粗暴的总数量限流，而不是平均速率限流。

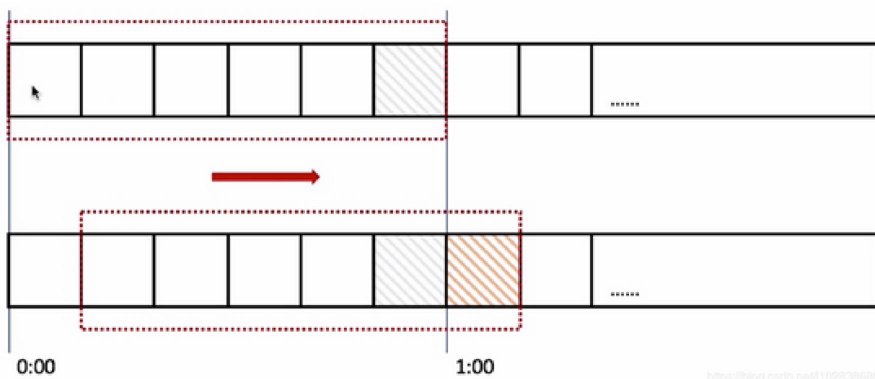


这个方法有一个致命问题：临界问题——当遇到恶意请求，在0:59时，瞬间请求100次，并且在1:00请求100次，那么这个用户在1秒内请求了200次，用户可以在重置节点突发请求，而瞬间超过我们设置的速率限制，用户可能通过算法漏洞击垮我们的应用。



这个问题我们可以使用滑动窗口解决。

滑动窗口

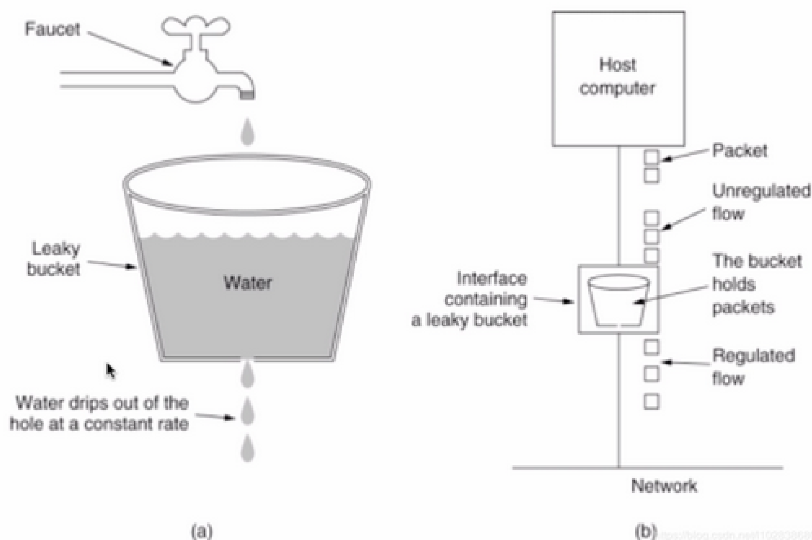


在上图中，整个红色矩形框是一个时间窗口，在我们的例子中，一个时间窗口就是1分钟，然后我们将时间窗口进行划分，如上图我们把滑动窗口划分为6格，所以每一格代表10秒，每超过10秒，我们的时间窗口就会向右滑动一格，每一格都有自己独立的计数器，例如：一个请求在0:35到达， 那么0:30到0:39的计数器会+1，那么滑动窗口是怎么解决临界点的问题呢？如上图，0:59到达的100个请求会在灰色区域格子中，而1：00到达的请求会在红色格子中，窗口会向右滑动一格，那么此时间窗口内的总请求数共200个，超过了限定的100，所以此时能够检测出来触发了限流。回头看看计数器算法，会发现，其实计数器算法就是窗口滑动算法，只不过计数器算法没有对时间窗口进行划分，所以是一格。

由此可见，当滑动窗口的格子划分越多，限流的统计就会越精确。

漏桶算法

算法的思路就是水（请求）先进入到漏桶里面，漏桶以恒定的速度流出，当水流的速度过大就会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。如下图所示。



漏桶算法不支持突发流量。

令牌桶算法

从上图中可以看出，令牌算法有点复杂，桶里存放着令牌token。桶一开始是空的，token以固定的速率r往桶里面填充，直到达到桶的容量，多余的token会被丢弃。每当一个请求过来时，就会尝试着移除一个token，如果没有token，请求无法通过。

令牌桶算法支持突发流量。

令牌桶算法实现

Guava框架提供了令牌桶算法的实现，可直接使用这个框架的RateLimiter类创建一个令牌桶限流器，比如：每秒放置的令牌桶的数量为5，那么RateLimiter对象可以保证1秒内不会放入超过5个令牌，并且以固定速率进行放置令牌，达到平滑输出的效果。

平滑流量示例

这里，我写了一个使用Guava框架实现令牌桶算法的示例，如下所示。

```
package io.binghe.limit.guava;

import com.google.common.util.concurrent.RateLimiter;

/**
 * @author binghe
 * @version 1.0.0
 * @description 令牌桶算法
 */
public class TokenBucketLimiter {
    public static void main(String[] args){
        //每秒钟生成5个令牌
        RateLimiter limiter = RateLimiter.create(5);

        //返回值表示从令牌桶中获取一个令牌所花费的时间，单位是秒
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
        System.out.println(limiter.acquire(1));
    }
}
```

代码的实现非常简单，就是使用Guava框架的RateLimiter类生成了一个每秒向桶中放入5个令牌的对象，然后不断从桶中获取令牌。我们先来运行下这段代码，输出的结果信息如下所示。

```
0.0
0.197294
0.191278
0.19997
0.199305
0.200472
0.200184
0.199417
0.200111
0.199759
```

从输出结果可以看出：第一次从桶中获取令牌时，返回的时间为0.0，也就是没耗费时间。之后每次从桶中获取令牌时，都会耗费一定的时间，这是为什么呢？按理说，向桶中放入了5个令牌后，再从桶中获取令牌也应该和第一次一样并不会花费时间啊！

因为在Guava的实现是这样的：我们使用`RateLimiter.create(5)`创建令牌桶对象时，表示每秒新增5个令牌，1秒等于1000毫秒，也就是每隔200毫秒向桶中放入一个令牌。

当我们运行程序时，程序运行到`RateLimiter limiter = RateLimiter.create(5);`时，就会向桶中放入一个令牌，当程序运行到第一个`System.out.println(limiter.acquire(1));`时，由于桶中已经存在一个令牌，直接获取这个令牌，并没有花费时间。然而程序继续向下执行时，由于程序会每隔200毫秒向桶中放入一个令牌，所以，获取令牌时，花费的时间几乎都是200毫秒左右。

突发流量示例

我们再来看一个突发流量的示例，代码示例如下所示。

```java
package io.binghe.limit.guava;

import com.google.common.util.concurrent.RateLimiter;

/**
 * @author binghe
 * @version 1.0.0
 * @description 令牌桶算法
 */
public class TokenBucketLimiter {
    public static void main(String[] args){
        //每秒钟生成5个令牌
        RateLimiter limiter = RateLimiter.create(5);

        //返回值表示从令牌桶中获取一个令牌所花费的时间，单位是秒
        System.out.println(limiter.acquire(50));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
        System.out.println(limiter.acquire(5));
    }
}
```

上述代码表示的含义为：每秒向桶中放入5个令牌，第一次从桶中获取50个令牌，也就是我们说的突发流量，后续每次从桶中获取5个令牌。接下来，我们运行上述代码看下效果。

```
0.0
9.998409
0.99109
1.000148
0.999752
```

运行代码时，会发现当命令行打印出0.0后，会等很久才会打印出后面的输出结果。

程序每秒钟向桶中放入5个令牌，当程序运行到 `RateLimiter limiter = RateLimiter.create(5);` 时，就会向桶中放入令牌。当运行到 `System.out.println(limiter.acquire(50));` 时，发现很快就会获取到令牌，花费了0.0秒。接下来，运行到第一个`System.out.println(limiter.acquire(5));`时，花费了9.998409秒。小伙们可以思考下，为什么这里会花费10秒中的时间呢？

这是因为我们使用`RateLimiter limiter = RateLimiter.create(5);`代码向桶中放入令牌时，一秒钟放入5个，而`System.out.println(limiter.acquire(50));`需要获取50个令牌，也就是获取50个令牌需要花费10秒钟时间，这是因为程序向桶中放入50个令牌需要10秒钟。程序第一次从桶中获取令牌时，很快就获取到了。而第二次获取令牌时，花费了将近10秒的时间。

Guava框架支持突发流量，但是在突发流量之后再次请求时，会被限速，也就是说：在突发流量之后，再次请求时，会弥补处理突发请求所花费的时间。所以，我们的突发示例程序中，在一次从桶中获取50个令牌后，再次从桶中获取令牌，则会花费10秒左右的时间。

Guava令牌桶算法的特点

- RateLimiter使用令牌桶算法，会进行令牌的累积，如果获取令牌的频率比较低，则不会导致等待，直接获取令牌。
- RateLimiter由于会累积令牌，所以可以应对突发流量。也就是说如果同时请求5个令牌，由于此时令牌桶中有累积的令牌，能够快速响应请求。
- RateLimiter在没有足够的令牌发放时，采用的是滞后的方式进行处理，也就是前一个请求获取令牌所需要等待的时间由下一次请求来承受和弥补，也就是代替前一个请求进行等待。（这里，小伙伴们要好好理解下）

HTTP接口限流实战

这里，我们实现Web接口限流，具体方式为：使用自定义注解封装基于令牌桶限流算法实现接口限流。

不使用注解实现接口限流

搭建项目

这里，我们使用SpringBoot项目来搭建Http接口限流项目，SpringBoot项目本质上还是一个Maven项目。所以，小伙伴们可以直接创建一个Maven项目，我这里的项目名称为mykit-ratelimiter-test。接下来，在pom.xml文件中添加如下依赖使项目构建为一个SpringBoot项目。

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.6.RELEASE</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <groupId>io.mykit.limiter</groupId>
    <artifactId>mykit-ratelimiter-test</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>mykit-ratelimiter-test</name>

    <properties>
        <guava.version>28.2-jre</guava.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-undertow</artifactId>
        </dependency>
```

```xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>

        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>${guava.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version><!--$NO-MVN-MAN-VER$-->
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

可以看到，我在项目中除了引用了SpringBoot相关的Jar包外，还引用了guava框架，版本为28.2-jre。

创建核心类

这里，我主要是模拟一个支付接口的限流场景。首先，我们定义一个PayService接口和MessageService接口。PayService接口主要用于模拟后续的支付业务，MessageService接口模拟发送消息。接口的定义分别如下所示。

- PayService

```java
package io.mykit.limiter.service;
import java.math.BigDecimal;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟支付
 */
public interface PayService {
    int pay(BigDecimal price);
}
```

- MessageService

```java
package io.mykit.limiter.service;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟发送消息服务
 */
public interface MessageService {
    boolean sendMessage(String message);
}
```

接下来，创建二者的实现类，分别如下。

- MessageServiceImpl

```java
package io.mykit.limiter.service.impl;
import io.mykit.limiter.service.MessageService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟实现发送消息
 */
@Service
public class MessageServiceImpl implements MessageService {
    private final Logger logger = LoggerFactory.getLogger(MessageServiceImpl.class);
    @Override
    public boolean sendMessage(String message) {
        logger.info("发送消息成功===>>" + message);
        return true;
    }
}
```

- PayServiceImpl

```java
package io.mykit.limiter.service.impl;
import io.mykit.limiter.service.PayService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import java.math.BigDecimal;
/**
 * @author binghe
 * @version 1.0.0
 * @description 模拟支付
 */
@Service
public class PayServiceImpl implements PayService {
    private final Logger logger = LoggerFactory.getLogger(PayServiceImpl.class);
    @Override
    public int pay(BigDecimal price) {
        logger.info("支付成功===>>" + price);
        return 1;
    }
}
```

由于是模拟支付和发送消息，所以，我在具体实现的方法中打印出了相关的日志，并没有实现具体的业务逻辑。

接下来，就是创建我们的Controller类PayController，在PayController类的接口pay()方法中使用了限流，每秒钟向桶中放入2个令牌，并且客户端从桶中获取令牌，如果在500毫秒内没有获取到令牌的话，我们可以则直接走服务降级处理。

PayController的代码如下所示。

```java
package io.mykit.limiter.controller;
import com.google.common.util.concurrent.RateLimiter;
import io.mykit.limiter.service.MessageService;
import io.mykit.limiter.service.PayService;
import org.slf4j.Logger;
```

```java
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.math.BigDecimal;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试接口限流
 */
@RestController
public class PayController {
    private final Logger logger = LoggerFactory.getLogger(PayController.class);
    /**
     * RateLimiter的create()方法中传入一个参数，表示以固定的速率2r/s，即以每秒2个令牌的速率向桶中放入令牌
     */
    private RateLimiter rateLimiter = RateLimiter.create(2);

    @Autowired
    private MessageService messageService;
    @Autowired
    private PayService payService;
    @RequestMapping("/boot/pay")
    public String pay(){
        //记录返回接口
        String result = "";
        //限流处理，客户端请求从桶中获取令牌，如果在500毫秒没有获取到令牌，则直接走服务降级处理
        boolean tryAcquire = rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS);
        if (!tryAcquire){
            result = "请求过多，降级处理";
            logger.info(result);
            return result;
        }
        int ret = payService.pay(BigDecimal.valueOf(100.0));
        if(ret > 0){
            result = "支付成功";
            return result;
        }
        result = "支付失败，再试一次吧...";
        return result;
    }
}
```

最后，我们来创建mykit-ratelimiter-test项目的核心启动类，如下所示。

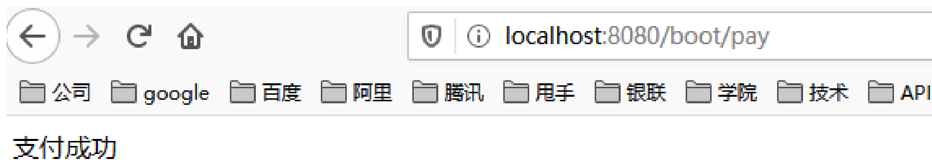
```java
package io.mykit.limiter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author binghe
 * @version 1.0.0
 * @description 项目启动类
 */
@SpringBootApplication
public class MykitLimiterApplication {

    public static void main(String[] args){
        SpringApplication.run(MykitLimiterApplication.class, args);
    }
}
```

至此，我们不使用注解方式实现限流的Web应用就基本完成了。

运行项目

项目创建完成后，我们来运行项目，运行SpringBoot项目比较简单，直接运行MykitLimiterApplication类的main()方法即可。

项目运行成功后，我们在浏览器地址栏输入链接：http://localhost:8080/boot/pay。页面会输出"支付成功"的字样，说明项目搭建成功了。如下所示。



此时，我只访问了一次，并没有触发限流。接下来，我们不停的刷浏览器，此时，浏览器会输出"支付失败，再试一次吧..."的字样，如下所示。

在PayController类中还有一个sendMessage()方法，模拟的是发送消息的接口，同样使用了限流操作，具体代码如下所示。

```java
@RequestMapping("/boot/send/message")
public String sendMessage(){
    //记录返回接口
    String result = "";
    //限流处理，客户端请求从桶中获取令牌，如果在500毫秒没有获取到令牌，则直接走服务降级处理
    boolean tryAcquire = rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS);
    if (!tryAcquire){
        result = "请求过多，降级处理";
        logger.info(result);
        return result;
    }
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "消息发送成功";
        return result;
    }
    result = "消息发送失败，再试一次吧...";
    return result;
}
```

sendMessage()方法的代码逻辑和运行效果与pay()方法相同，我就不再浏览器访问 http://localhost:8080/boot/send/message 地址的访问效果了，小伙伴们可以自行验证。

不使用注解实现限流缺点

通过对项目的编写，我们可以发现，当在项目中对接口进行限流时，不使用注解进行开发，会导致代码出现大量冗余，每个方法中几乎都要写一段相同的限流逻辑，代码十分冗余。

如何解决代码冗余的问题呢？我们可以使用自定义注解进行实现。

使用注解实现接口限流

使用自定义注解，我们可以将一些通用的业务逻辑封装到注解的切面中，在需要添加注解业务逻辑的方法上加上相应的注解即可。针对我们这个限流的实例来说，可以基于自定义注解实现。

实现自定义注解

实现，我们来创建一个自定义注解，如下所示。

```java
package io.mykit.limiter.annotation;
import java.lang.annotation.*;
/**
 * @author binghe
 * @version 1.0.0
 * @description 实现限流的自定义注解
 */
```

```java
@Target(value = ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRateLimiter {
    //向令牌桶放入令牌的速率
    double rate();
    //从令牌桶获取令牌的超时时间
    long timeout() default 0;
}
```

自定义注解切面实现

接下来，我们还要实现一个切面类MyRateLimiterAspect，如下所示。

```java
package io.mykit.limiter.aspect;

import com.google.common.util.concurrent.RateLimiter;
import io.mykit.limiter.annotation.MyRateLimiter;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 一般限流切面类
 */
@Aspect
@Component
public class MyRateLimiterAspect {

    private RateLimiter rateLimiter = RateLimiter.create(2);

    @Pointcut("execution(public * io.mykit.limiter.controller.*.*(..))")
    public void pointcut(){

    }

    /**
     * 核心切面方法
     */
    @Around("pointcut()")
    public Object process(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
        MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();

        //使用反射获取方法上是否存在@MyRateLimiter注解
        MyRateLimiter myRateLimiter = signature.getMethod().getDeclaredAnnotation(MyRateLimiter.class);
        if(myRateLimiter == null){
            //程序正常执行，执行目标方法
            return proceedingJoinPoint.proceed();
        }
        //获取注解上的参数
        //获取配置的速率
        double rate = myRateLimiter.rate();
        //获取客户端等待令牌的时间
        long timeout = myRateLimiter.timeout();

        //设置限流速率
        rateLimiter.setRate(rate);

        //判断客户端获取令牌是否超时
```

```java
        boolean tryAcquire = rateLimiter.tryAcquire(timeout, TimeUnit.MILLISECONDS);
        if(!tryAcquire){
            //服务降级
            fullback();
            return null;
        }
        //获取到令牌，直接执行
        return proceedingJoinPoint.proceed();

    }

    /**
     * 降级处理
     */
    private void fullback() {
        response.setHeader("Content-type", "text/html;charset=UTF-8");
        PrintWriter writer = null;
        try {
            writer =  response.getWriter();
            writer.println("出错了，重试一次试试？");
            writer.flush();;
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(writer != null){
                writer.close();
            }
        }
    }
}
```

自定义切面的功能比较简单，我就不细说了，大家有啥问题可以关注【冰河技术】微信公众号来进行提问。

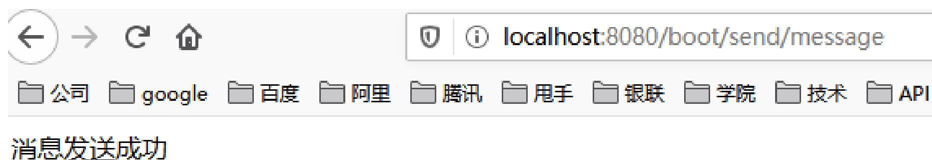
接下来，我们改造下PayController类中的sendMessage()方法，修改后的方法片段代码如下所示。

```java
@MyRateLimiter(rate = 1.0, timeout = 500)
@RequestMapping("/boot/send/message")
public String sendMessage(){
    //记录返回接口
    String result = "";
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "消息发送成功";
        return result;
    }
    result = "消息发送失败，再试一次吧...";
    return result;
}
```

运行部署项目

部署项目比较简单，只需要运行MykitLimiterApplication类下的main()方法即可。这里，为了简单，我们还是从浏览器中直接输入链接地址来进行访问

效果如下所示。



接下来，我们不断的刷新浏览器。会出现"消息发送失败，再试一次吧.."的字样，说明已经触发限流操作。

基于限流算法实现限流的缺点

上面介绍的限流方式都只能用于单机部署的环境中，如果将应用部署到多台服务器进行分布式、集群，则上面限流的方式就不适用了，此时，我们需要使用分布式限流。至于在分布式场景下，如何实现限流操作，我们就在下一篇中进行介绍。

分布式限流实战

前面介绍的限流方案有一个缺陷就是：它不是全局的，不是分布式的，无法很好的应对分布式场景下的大流量冲击。那么，接下来，我们就介绍下如何实现亿级流量下的分布式限流。

分布式限流的关键就是需要将限流服务做成全局的，统一的。可以采用Redis+Lua技术实现，通过这种技术可以实现高并发和高性能的限流。

Lua是一种轻量小巧的脚本编程语言，用标准的C语言编写的开源脚本，其设计的目的是为了嵌入到应用程序中，为应用程序提供灵活的扩展和定制功能。

Redis+Lua脚本实现分布式限流思路

我们可以使用Redia+Lua脚本的方式来对我们的分布式系统进行统一的全局限流，Redis+Lua实现的Lua脚本：

```
local key = KEYS[1]  --限流KEY(一秒一个)
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
    return 0
else --请求数+1，并设置2秒过期
    redis.call("INCRBY", key, "1")
    redis.call("expire", key "2")
    return 1
end
```

我们可以按照如下的思路来理解上述Lua脚本代码。

（1）在Lua脚本中，有两个全局变量，用来接收Redis应用端传递的键和其他参数，分别为：KEYS、ARGV；

（2）在应用端传递KEYS时是一个数组列表，在Lua脚本中通过索引下标方式获取数组内的值。

（3）在应用端传递ARGV时参数比较灵活，可以是一个或多个独立的参数，但对应到Lua脚本中统一用ARGV这个数组接收，获取方式也是通过数组下标获取。

（4）以上操作是在一个Lua脚本中，又因为我当前使用的是Redis 5.0版本（Redis 6.0支持多线程），执行的请求是单线程的，因此，Redis+Lua的处理方式是线程安全的，并且具有原子性。

这里，需要注意一个知识点，那就是原子性操作：如果一个操作时不可分割的，是多线程安全的，我们就称为原子性操作。

接下来，我们可以使用如下Java代码来判断是否需要限流。

```
//List设置Lua的KEYS[1]
String key = "ip:" + System.currentTimeMillis() / 1000;
List<String> keyList = Lists.newArrayList(key);

//List设置Lua的ARGV[1]
List<String> argvList = Lists.newArrayList(String.valueOf(value));

//调用Lua脚本并执行
List result = stringRedisTemplate.execute(redisScript, keyList, argvList)
```

至此，我们简单的介绍了使用Redis+Lua脚本实现分布式限流的总体思路，并给出了Lua脚本的核心代码和Java程序调用Lua脚本的核心代码。接下来，我们就动手写一个使用Redis+Lua脚本实现的分布式限流案例。

Redis+Lua脚本实现分布式限流案例

这里，我们和在HTTP接口限流实战的实现方式类似，也是通过自定义注解的形式来实现分布式、大流量场景下的限流，只不过这里我们使用了Redis+Lua脚本的方式实现了全局统一的限流模式。接下来，我们就一起手动实现这个案例。

创建注解

首先，我们在项目中，定义个名称为MyRedisLimiter的注解，具体代码如下所示。

```
package io.mykit.limiter.annotation;
import org.springframework.core.annotation.AliasFor;
```

```java
import java.lang.annotation.*;
/**
 * @author binghe
 * @version 1.0.0
 * @description 自定义注解实现分布式限流
 */
@Target(value = ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRedisLimiter {
    @AliasFor("limit")
    double value() default Double.MAX_VALUE;
    double limit() default Double.MAX_VALUE;
}
```

在MyRedisLimiter注解内部，我们为value属性添加了别名limit，在我们真正使用@MyRedisLimiter注解时，即可以使用@MyRedisLimiter(10)，也可以使用@MyRedisLimiter(value=10)，还可以使用@MyRedisLimiter(limit=10)。

创建切面类

创建注解后，我们就来创建一个切面类MyRedisLimiterAspect，MyRedisLimiterAspect类的作用主要是解析@MyRedisLimiter注解，并且执行限流的规则。这样，就不需要我们在每个需要限流的方法中执行具体的限流逻辑了，只需要我们在需要限流的方法上添加@MyRedisLimiter注解即可，具体代码如下所示。

```java
package io.mykit.limiter.aspect;
import com.google.common.collect.Lists;
import io.mykit.limiter.annotation.MyRedisLimiter;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.scripting.support.ResourceScriptSource;
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.util.List;

/**
 * @author binghe
 * @version 1.0.0
 * @description MyRedisLimiter注解的切面类
 */
@Aspect
@Component
public class MyRedisLimiterAspect {
    private final Logger logger = LoggerFactory.getLogger(MyRedisLimiter.class);
    @Autowired
    private HttpServletResponse response;
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private DefaultRedisScript<List> redisScript;

    @PostConstruct
    public void init(){
        redisScript = new DefaultRedisScript<List>();
        redisScript.setResultType(List.class);
        redisScript.setScriptSource(new ResourceScriptSource(new ClassPathResource(("limit.lua"))));
    }

    @Pointcut("execution(public * io.mykit.limiter.controller.*.*(..))")
    public void pointcut(){
```

```java
    }

    @Around("pointcut()")
    public Object process(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{
        MethodSignature signature = (MethodSignature) proceedingJoinPoint.getSignature();
        //使用反射获取MyRedisLimiter注解
        MyRedisLimiter myRedisLimiter =
signature.getMethod().getDeclaredAnnotation(MyRedisLimiter.class);
        if(myRedisLimiter == null){
            //正常执行方法
            return proceedingJoinPoint.proceed();
        }
        //获取注解上的参数，获取配置的速率
        double value = myRedisLimiter.value();
        //List设置Lua的KEYS[1]
        String key = "ip:" + System.currentTimeMillis() / 1000;
        List<String> keyList = Lists.newArrayList(key);

        //List设置Lua的ARGV[1]
        List<String> argvList = Lists.newArrayList(String.valueOf(value));

        //调用Lua脚本并执行
        List result = stringRedisTemplate.execute(redisScript, keyList, String.valueOf(value));
        logger.info("Lua脚本的执行结果：" + result);

        //Lua脚本返回0，表示超出流量大小，返回1表示没有超出流量大小。
        if("0".equals(result.get(0).toString())){
            fullBack();
            return null;
        }

        //获取到令牌，继续向下执行
        return proceedingJoinPoint.proceed();
    }

    private void fullBack() {
        response.setHeader("Content-Type" ,"text/html;charset=UTF8");
        PrintWriter writer = null;
        try{
            writer = response.getWriter();
            writer.println("回退失败，请稍后阅读。。。");
            writer.flush();
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            if(writer != null){
                writer.close();
            }
        }
    }
}
```

上述代码会读取项目classpath目录下的limit.lua脚本文件来确定是否执行限流的操作，调用limit.lua文件执行的结果返回0则表示执行限流逻辑，否则不执行限流逻辑。既然，项目中需要使用Lua脚本，那么，接下来，我们就需要在项目中创建Lua脚本。

创建limit.lua脚本文件

在项目的classpath目录下创建limit.lua脚本文件，文件的内容如下所示。

```lua
local key = KEYS[1]  --限流KEY(一秒一个)
local limit = tonumber(ARGV[1]) --限流大小
local current = tonumber(redis.call('get', key) or "0")
if current + 1 > limit then --如果超出限流大小
    return 0
else --请求数+1，并设置2秒过期
    redis.call("INCRBY", key, "1")
    redis.call("expire", key "2")
    return 1
end
```

limit.lua脚本文件的内容比较简单，这里就不再赘述了。

接口添加注解

注解类、解析注解的切面类、Lua脚本文件都已经准备好。那么，接下来，我们在PayController类中在sendMessage2()方法上添加@MyRedisLimiter注解，并且将limit属性设置为10，如下所示。

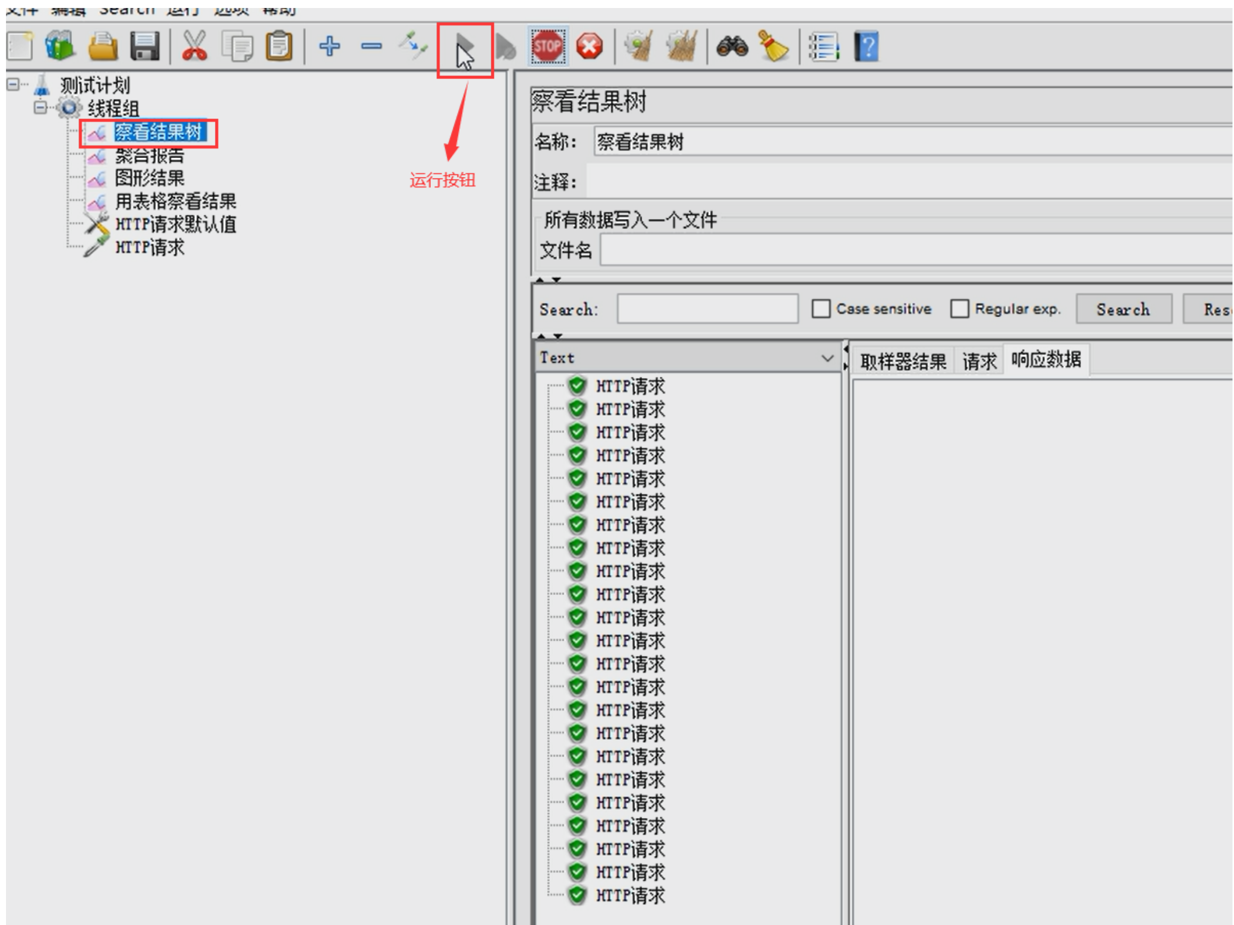
```
@MyRedisLimiter(limit = 10)
@RequestMapping("/boot/send/message2")
public String sendMessage2(){
    //记录返回接口
    String result = "";
    boolean flag = messageService.sendMessage("恭喜您成长值+1");
    if (flag){
        result = "短信发送成功！";
        return result;
    }
    result = "哎呀，服务器开小差了，请再试一下吧";
    return result;
}
```

此处，我们限制了sendMessage2()方法，每秒钟最多只能处理10个请求。那么。接下来，我们就使用JMeter对sendMessage2()进行测试。

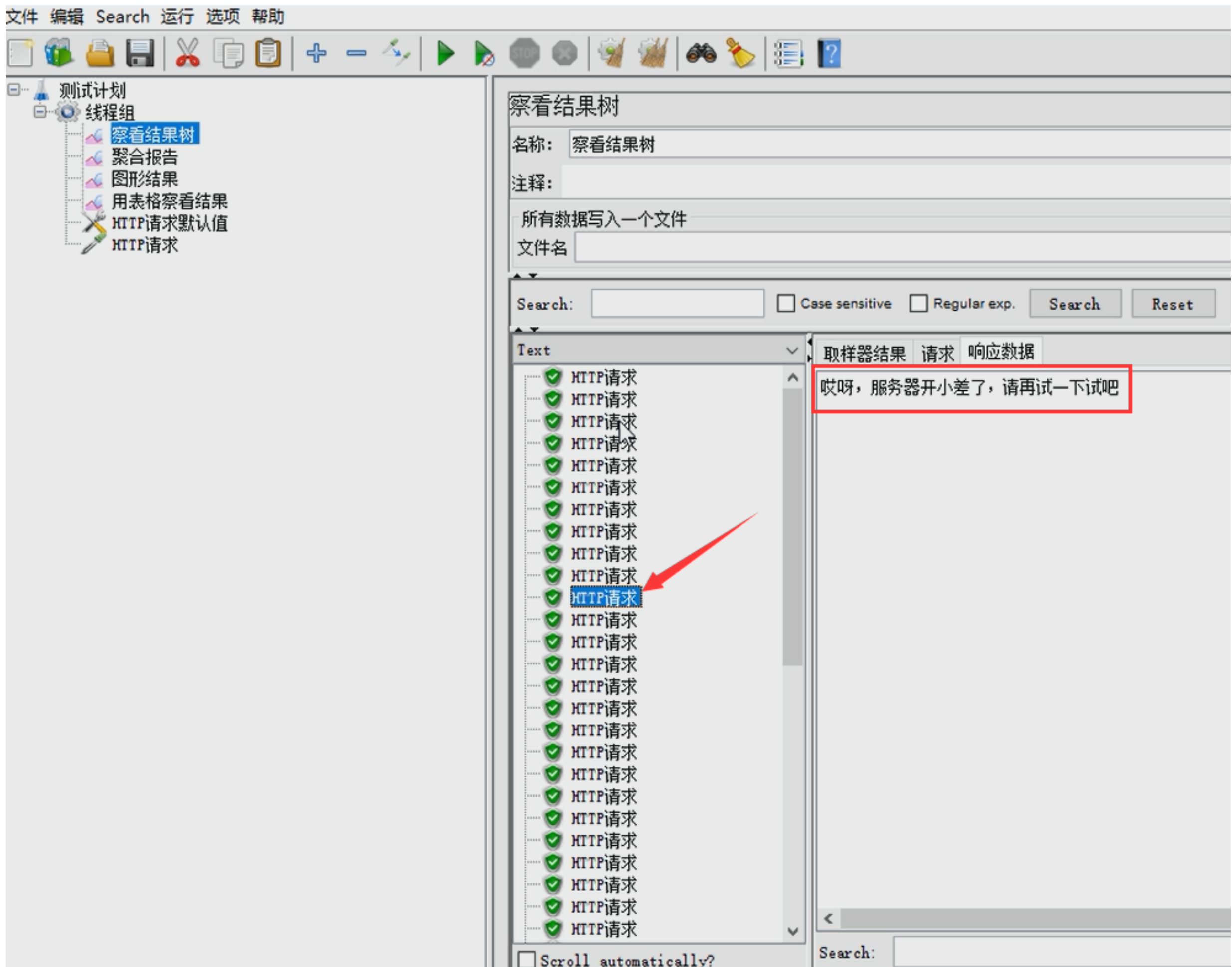
测试分布式限流

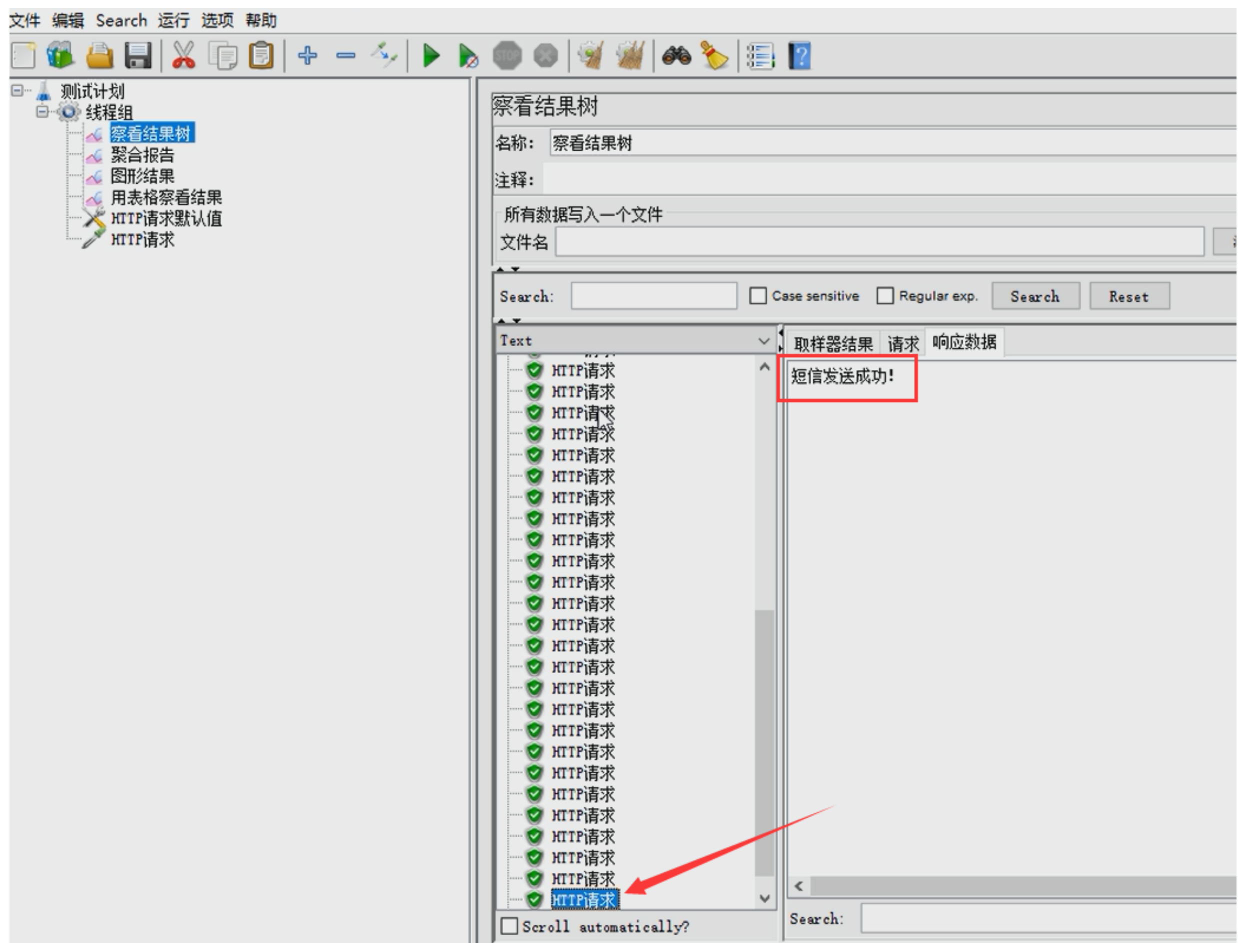
此时，我们使用JMeter进行压测，这里，我们配置的线程数为50，也就是说：会有50个线程同时访问我们写的接口。JMeter的配置如下所示。

保存并运行Jemeter，如下所示。



运行完成后，我们来查看下JMeter的测试结果，如下所示。





从测试结果可以看出，测试中途有部分接口的访问返回了"哎呀，服务器开小差了，请再试一下吧"，说明接口被限流了。而再往后，又有部分接口成功返回了"短信发送成功！"的字样。这是因为我们设置的是接口每秒最多接受10次请求，在第一秒内访问接口时，前面的10次请求成功返回"短信发送成功！"的字样，后面再访问接口就会返回"哎呀，服务器开小差了，请再试一下吧"。而后面的请求又返回了"短信发送成功！"的字样，说明后面的请求已经是在第二秒的时候调用的接口。

我们使用Redis+Lua脚本的方式实现的限流方式，可以将Java程序进行集群部署，这种方式实现的是全局的统一的限流，无论客户端访问的是集群中的哪个节点，都会对访问进行计数并实现最终的限流效果。

这种思想就有点像分布式锁了，小伙伴们可以关注【冰河技术】微信公众号阅读我写的一篇《【高并发】高并发分布式锁架构解密，不是所有的锁都是分布式锁！！》来深入理解如何实现真正线程安全的分布式锁，此文章，以循序渐进的方式深入剖析了实现分布式锁过程中的各种坑和解决方案，让你真正理解什么才是分布式锁。

Nginx+Lua实现分布式限流

Nginx+Lua实现分布式限流，通常会用在应用的入口处，也就是对系统的流量入口进行限流。这里，我们也以一个实际案例的形式来说明如何使用Nginx+Lua来实现分布式限流。

首先，我们需要创建一个Lua脚本，脚本文件的内容如下所示。

```lua
local locks = require "resty.lock"

local function acquire()
    local lock =locks:new("locks")
    local elapsed, err =lock:lock("limit_key") --互斥锁
    local limit_counter =ngx.shared.limit_counter --计数器

    local key = "ip:" ..os.time()
    local limit = 5 --限流大小
    local current =limit_counter:get(key)

    if current ~= nil and current + 1> limit then --如果超出限流大小
       lock:unlock()
       return 0
    end
    if current == nil then
       limit_counter:set(key, 1, 1) --第一次需要设置过期时间，设置key的值为1，过期时间为1秒
    else
        limit_counter:incr(key, 1) --第二次开始加1即可
```

```
    end
    lock:unlock()
    return 1
end
ngx.print(acquire())
```

实现中我们需要使用lua-resty-lock互斥锁模块来解决原子性问题(在实际工程中使用时请考虑获取锁的超时问题)，并使用ngx.shared.DICT共享字典来实现计数器。如果需要限流则返回0，否则返回1。使用时需要先定义两个共享字典（分别用来存放锁和计数器数据）。

接下来，需要在Nginx的nginx.conf配置文件中定义数据字典，如下所示。

```
http {
    ……
    lua_shared_dict locks 10m;
    lua_shared_dict limit_counter 10m;
}
```

灵魂拷问

说到这里，相信有很多小伙伴可能会问：如果应用并发量非常大，那么，Redis或者Nginx能不能扛的住呢？

可以这么说：Redis和Nginx基本都是高性能的互联网组件，对于一般互联网公司的高并发流量是完全没有问题的。为什么这么说呢？咱们继续往下看。

如果你的应用流量真的非常大，可以通过一致性哈希将分布式限流进行分片，还可以将限流降级为应用级限流；解决方案也非常多，可以根据实际情况进行调整，使用Redis+Lua的方式进行限流，是可以稳定达到对上亿级别的高并发流量进行限流的（笔者亲身经历）。

需要注意的是：面对高并发系统，尤其是这种流量上千万、上亿级别的高并发系统，我们不可能只用限流这一招，还要加上其他的一些措施，

对于分布式限流，目前遇到的场景是业务上的限流，而不是流量入口的限流。对于流量入口的限流，应该在接入层来完成。

对于秒杀场景来说，可以在流量入口处进行限流，小伙伴们可以关注【冰河技术】微信公众号，来阅读我写的《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》一文，来深入理解如何架构一个高并发秒杀系统

面试篇

面试官：讲讲高并发场景下如何优化加锁方式？

写在前面

> 很多时候，我们在并发编程中，涉及到加锁操作时，对代码块的加锁操作真的合理吗？还有没有需要优化的地方呢？

前言

在《【高并发】优化加锁方式时竟然死锁了！！》一文中，我们介绍了产生死锁时的四个必要条件，只有四个条件同时具备时才能发生死锁。其中，我们在**阻止请求与保持条件**时，采用了一次性申请所有的资源的方式。例如在我们完成转账操作的过程中，我们一次性申请账户A和账户B，两个账户都申请成功后，再执行转账的操作。其中，在我们实现的转账方法中，使用了死循环来循环获取资源，直到同时获取到账户A和账户B为止，核心代码如下所示。

```
//一次申请转出账户和转入账户，直到成功
while(!requester.applyResources(this, target)){
    //循环体为空
    ;
}
```

如果ResourcesRequester类的applyResources()方法执行的时间非常短，并且程序并发带来的冲突不大，程序循环几次到几十次就可以同时获取到转出账户和转入账户，这种方案就是可行的。

但是，如果ResourcesRequester类的applyResources()方法执行的时间比较长，或者说，程序并发带来的冲突比较大，此时，可能需要循环成千上万次才能同时获取到转出账户和转入账户。这样就太消耗CPU资源了，此时，这种方案就是不可行的了。

那么，有没有什么方式对这种方案进行优化呢？

问题分析

既然使用死循环一直获取资源这种方案存在问题，那我们换位思考一下。当线程执行时，发现条件不满足，是不是可以让线程进入等待状态？当条件满足的时候，通知等待的线程重新执行？

也就是说，如果线程需要的条件不满足，我们就让线程进入等待状态；如果线程需要的条件满足时，我们再通知等待的线程重新执行。这样，就能够避免程序进行循环等待进而消耗CPU的问题。

那么，问题又来了！当条件不满足时，如何实现让线程等待？当条件满足时，又如何唤醒线程呢？

不错，这是个问题！不过这个问题解决起来也非常简单。简单的说，就是**使用线程的等待与通知机制。**

线程的等待与通知机制

我们可以使用线程的等待与通知机制来优化**阻止请求与保持条件**时，循环获取账户资源的问题。具体的等待与通知机制如下所示。

执行的线程首先获取互斥锁，如果线程继续执行时，需要的条件不满足，则释放互斥锁，并进入等待状态；当线程继续执行需要的条件满足时，就通知等待的线程，重新获取互斥锁。

那么，说了这么多，Java支持这种线程的等待与通知机制吗？其实，这个问题问的就有点废话了，Java这么优秀（牛逼）的语言肯定支持啊，而且实现起来也比较简单。

用Java实现线程的等待与通知机制

实现方式

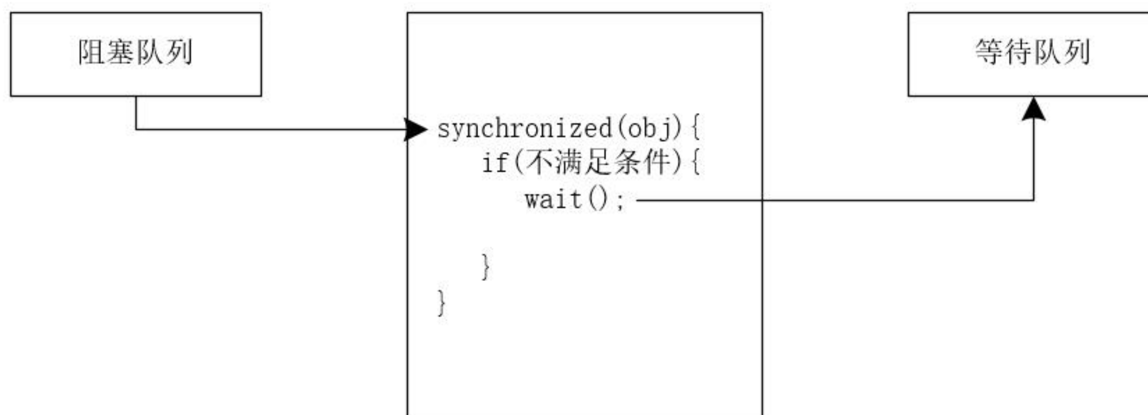
其实，使用Java语言实现线程的等待与通知机制有多种方式，这里我就简单的列举一种方式，其他的方式大家可以自行思考和实现，有不懂的地方也可以问我！

在Java语言中，实现线程的等待与通知机制，一种简单的方式就是使用synchronized并结合wait()、notify()和notifyAll()方法来实现。

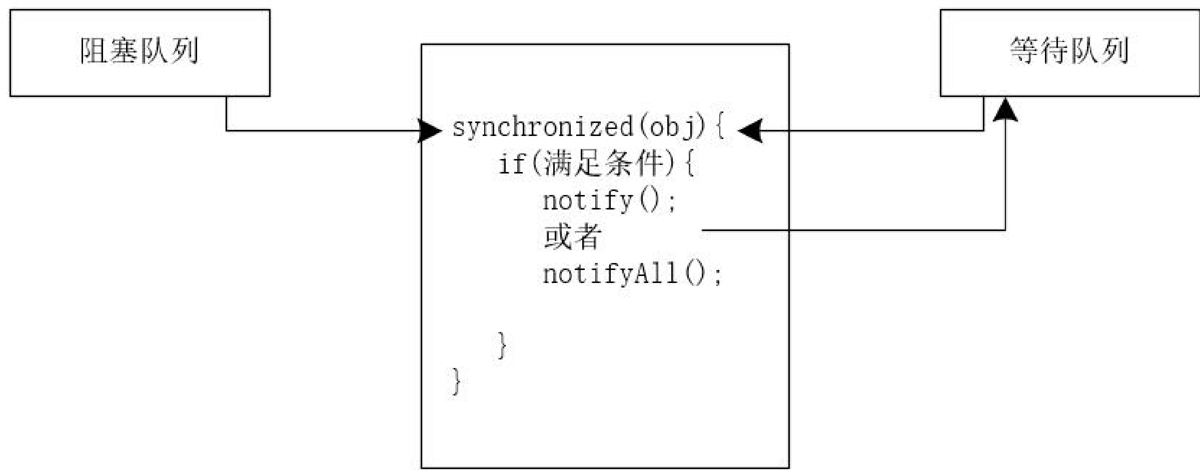
实现原理

我们使用synchronized加锁时，只允许一个线程进入synchronized保护的代码块，也就是临界区。如果一个线程进入了临界区，则其他的线程会进入阻塞队列里等待，这个阻塞队列和synchronized互斥锁是一对一的关系，也就是说，一把互斥锁对应着一个独立的阻塞队列。

在并发编程中，如果一个线程获得了synchronized互斥锁，但是不满足继续向下执行的条件，则需要进入等待状态。此时，可以使用Java中的wait()方法来实现。当调用wait()方法后，当前线程就会被阻塞，并且会进入一个等待队列中进行等待，这个由于调用wait()方法而进入的等待队列也是互斥锁的等待队列。而且，线程在进入等待队列的同时，会释放自身获得的互斥锁，这样，其他线程就有机会获得互斥锁，进而进入临界区了。整个过程可以表示成下图所示。



当线程执行的条件满足时，可以使用Java提供的notify()和notifyAll()方法来通知互斥锁等待队列中的线程，我们可以使用下图来简单的表示这个过程。



这里，需要注意如下事项：

（1）使用notify()和notifyAll()方法通知线程时，调用notify()和notifyAll()方法时，满足线程的执行条件，但是当线程真正执行的时候，条件可能已经不再满足了，可能有其他线程已经进入临界区执行。

（2）被通知的线程继续执行时，需要先获取互斥锁，因为在调用wait()方法等待时已经释放了互斥锁。

（3）wait()、notify()和notifyAll()方法操作的队列是互斥锁的等待队列，如果synchronized锁定的是this对象，则一定要使用this.wait()、this.notify()和this.notifyAll()方法；如果synchronized锁定的是target对象，则一定要使用target.wait()、target.notify()和target.notifyAll()方法。

（4）wait()、notify()和notifyAll()方法调用的前提是已经获取了相应的互斥锁，也就是说，wait()、notify()和notifyAll()方法都是在synchronized方法中或代码块中调用的。如果在synchronized方法外或代码块外调用了三个方法，或者锁定的对象是this，使用target对象调用三个方法的话，JVM会抛出java.lang.IllegalMonitorStateException异常。

具体实现

实现逻辑

在实现之前，我们还需要考虑以下几个问题：

- **选择哪个互斥锁**

在之前的程序中，我们在TansferAccount类中，存在一个ResourcesRequester 类的单例对象，所以，我们是可以使用this作为互斥锁的。这一点大家需要重点理解。

- **线程执行转账操作的条件**

转出账户和转入账户都没有被分配过。

- **线程什么时候进入等待状态**

线程继续执行需要的条件不满足的时候，进入等待状态。

- **什么时候通知等待的线程执行**

当存在线程释放账户的资源时，通知等待的线程继续执行。

综上，我们可以得出以下核心代码。

```
while(不满足条件){
    wait();
}
```

那么，问题来了！为何是在while循环中调用wait()方法呢？ 因为当wait()方法返回时，有可能线程执行的条件已经改变，也就是说，之前条件是满足的，但是现在已经不满足了，所以要重新检验条件是否满足。

实现代码

我们优化后的ResourcesRequester类的代码如下所示。

```
public class ResourcesRequester{
    //存放申请资源的集合
    private List<Object> resources = new ArrayList<Object>();
    //一次申请所有的资源
```

```java
    public synchronized void applyResources(Object source, Object target){
        while(resources.contains(source) || resources.contains(target)){
            try{
                wait();
            }catch(Exception e){
                e.printStackTrace();
            }
        }
        resources.add(source);
        resources.add(targer);
    }

    //释放资源
    public synchronized void releaseResources(Object source, Object target){
        resources.remove(source);
        resources.remove(target);
        notifyAll();
    }
}
```

生成ResourcesRequester单例对象的Holder类ResourcesRequesterHolder的代码如下所示。

```java
public class ResourcesRequesterHolder{
    private ResourcesRequesterHolder(){}

    public static ResourcesRequester getInstance(){
        return Singleton.INSTANCE.getInstance();
    }
    private enum Singleton{
        INSTANCE;
        private ResourcesRequester singleton;
        Singleton(){
            singleton = new ResourcesRequester();
        }
        public ResourcesRequester getInstance(){
            return singleton;
        }
    }
}
```

执行转账操作的类的代码如下所示。

```java
public class TansferAccount{
    //账户的余额
    private Integer balance;
    //ResourcesRequester类的单例对象
    private ResourcesRequester requester;

    public TansferAccount(Integer balance){
        this.balance = balance;
        this.requester = ResourcesRequesterHolder.getInstance();
    }
    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        //一次申请转出账户和转入账户，直到成功
        requester.applyResources(this, target))
        try{
            //对转出账户加锁
            synchronized(this){
                //对转入账户加锁
                synchronized(target){
                    if(this.balance >= transferMoney){
                        this.balance -= transferMoney;
                        target.balance += transferMoney;
                    }
                }
            }
        }finally{
            //最后释放账户资源
            requester.releaseResources(this, target);
```

```
            }
        }
    }
```

可以看到，我们在程序中通知处于等待状态的线程时，使用的是notifyAll()方法而不是notify()方法。**那notify()方法和notifyAll()方法两者有什么区别呢?**

notify()和notifyAll()的区别

- notify()方法

随机通知等待队列中的一个线程。

- notifyAll()方法

通知等待队列中的所有线程。

在实际工作过程中，如果没有特殊的要求，尽量使用notifyAll()方法。因为使用notify()方法是有风险的，可能会导致某些线程永久不会被通知到！

面试官：讲讲什么是缓存穿透？击穿？雪崩？如何解决？

写在前面

> 在前面的《【高并发】Redis如何助力高并发秒杀系统？看完这篇我彻底懂了！！》一文中，我们以高并发秒杀系统中扣减库存的场景为例，说明了Redis是如何助力秒杀系统的。那么，说到Redis，往往更多的场景是被用作系统的缓存，说到缓存，尤其是分布式缓存系统，在实际高并发场景下，稍有不慎，就会造成缓存穿透、缓存击穿和缓存雪崩的问题。那什么是缓存穿透？什么是缓存击穿，又什么是缓存雪崩呢？它们是如何造成的？又该如何解决呢？今天，我们就一起来探讨这些问题。

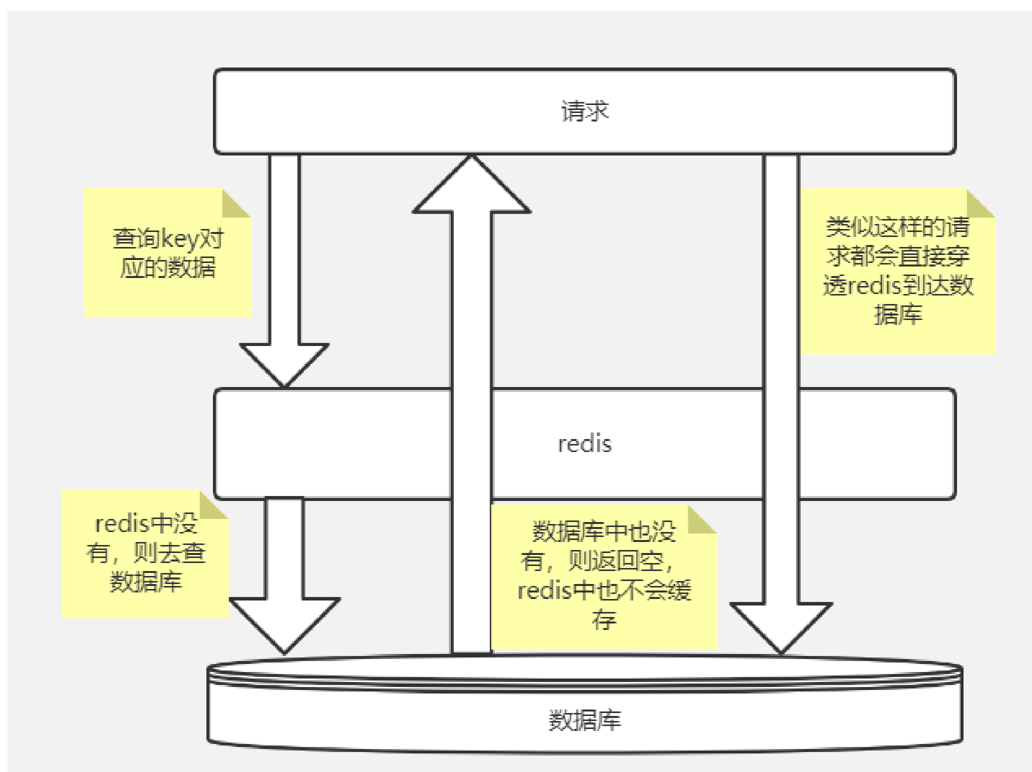
缓存穿透

首先，我们来说说缓存穿透。什么是缓存穿透呢？缓存穿透问题在一定程度上与缓存命中率有关。如果我们的缓存设计的不合理，缓存的命中率非常低，那么，数据访问的绝大部分压力都会集中在后端数据库层面。

什么是缓存穿透?

如果在请求数据时，在缓存层和数据库层都没有找到符合条件的数据，也就是说，在缓存层和数据库层都没有命中数据，那么，这种情况就叫作缓存穿透。

我们可以使用下图来表示缓存穿透的现象。



造成缓存穿透的主要原因就是：查询某个Key对应的数据，Redis缓存中没有相应的数据，则直接到数据库中查询。数据库中也不存在要查询的数据，则数据库会返回空，而Redis也不会缓存这个空结果。这就造成每次通过这样的Key去查询数据都会直接到数据库中查询，Redis不会缓存空结果。这就造成了缓存穿透的问题。

如何解决缓存穿透问题？

既然我们知道了造成缓存穿透的主要原因就是缓存中不存在相应的数据，直接到数据库查询，数据库返回空结果，缓存中不存储空结果。

那我们就自然而然的想到了第一种解决方案：就是把空对象缓存起来。当第一次从数据库中查询出来的结果为空时，我们就将这个空对象加载到缓存，并设置合理的过期时间，这样，就能够在一定程度上保障后端数据库的安全。

第二种解决缓存穿透问题的解决方案：就是使用布隆过滤器，布隆过滤器可以针对大数据量的、有规律的键值进行处理。一条记录是不是存在，本质上是一个Bool值，只需要使用 1bit 就可以存储。我们可以使用布隆过滤器将这种表示是、否等操作，压缩到一个数据结构中。比如，我们最熟悉的用户性别这种数据，就非常适合使用布隆过滤器来处理。

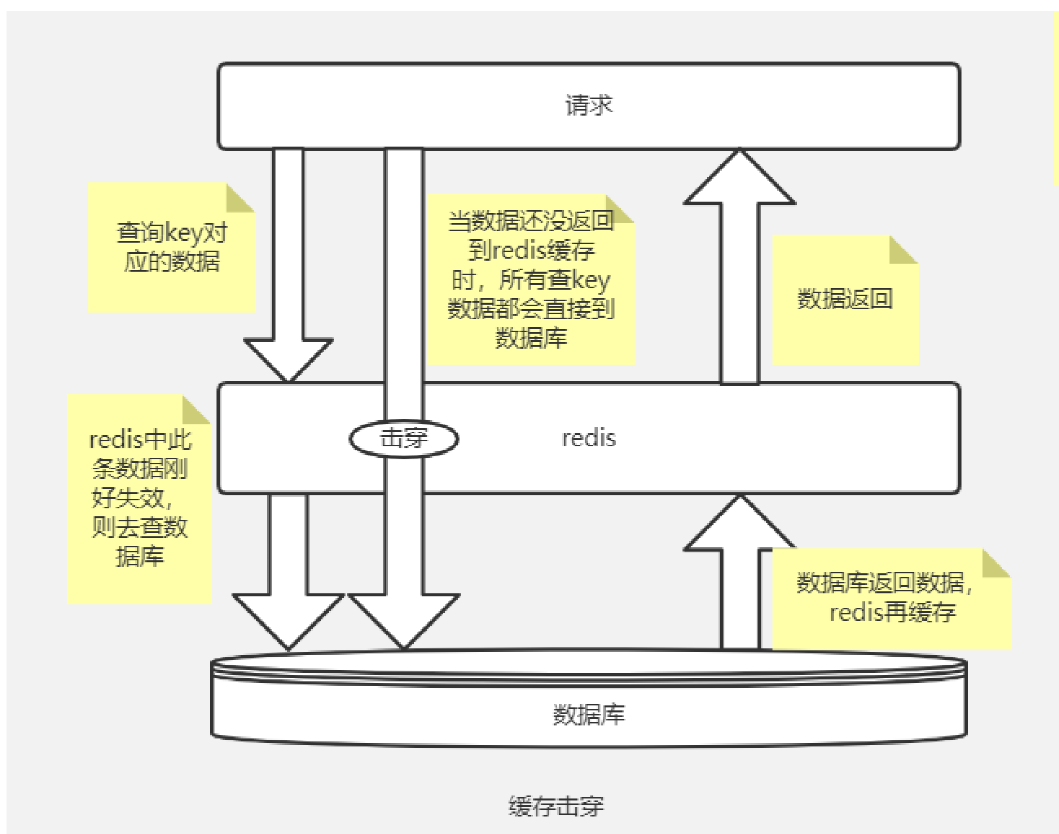
缓存击穿

如果我们为缓存中的大部分数据设置了相同的过期时间，则到了某一时刻，缓存中的数据就会批量过期。

什么是缓存击穿？

如果缓存中的数据在某个时刻批量过期，导致大部分用户的请求都会直接落在数据库上，这种现象就叫作缓存击穿。

我们可以使用下图来表示缓存击穿的线程。



缓存击穿

造成缓存击穿的主要原因就是：我们为缓存中的数据设置了过期时间。如果在某个时刻从数据库获取了大量的数据，并设置了相同的过期时间，这些缓存的数据就会在同一时刻失效，造成缓存击穿问题。

如何解决缓存击穿问题？

对于比较热点的数据，我们可以在缓存中设置这些数据永不过期；也可以在访问数据的时候，在缓存中更新这些数据的过期时间；如果是批量入库的缓存项，我们可以为这些缓存项分配比较合理的过期时间，避免同一时刻失效。

还有一种解决方案就是：使用分布式锁，保证对于每个Key同时只有一个线程去查询后端的服务，某个线程在查询后端服务的同时，其他线程没有获得分布式锁的权限，需要进行等待。不过在高并发场景下，这种解决方案对于分布式锁的访问压力比较大。

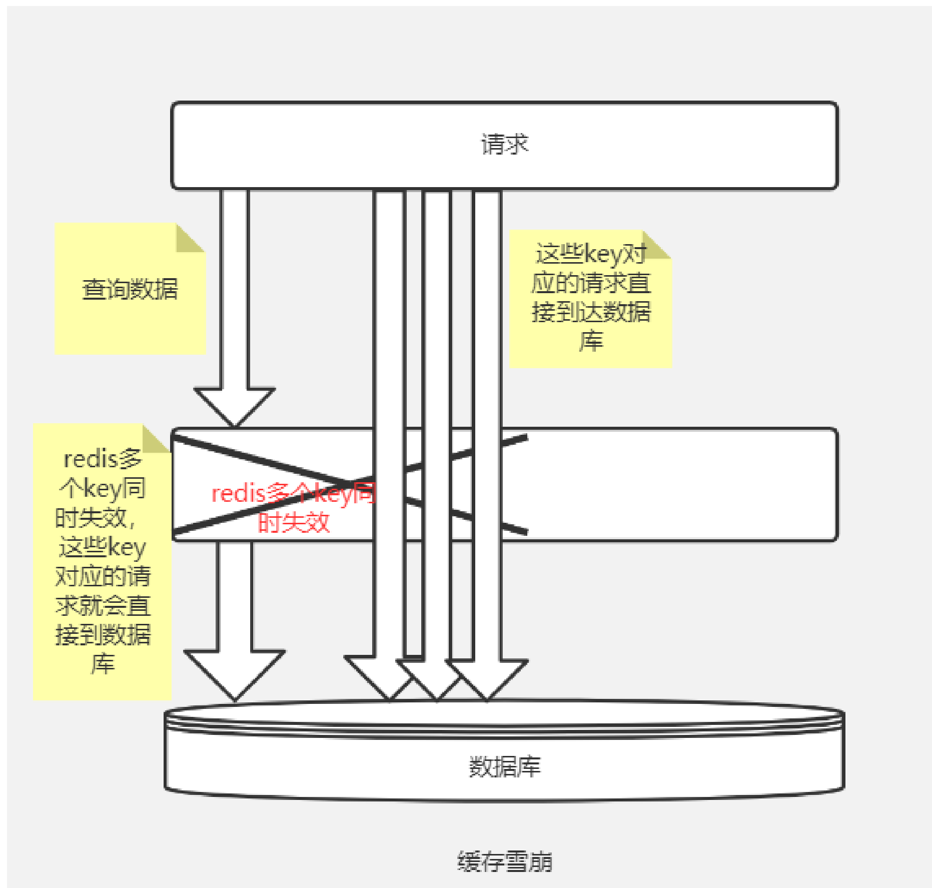
缓存雪崩

如果缓存系统出现故障，所有的并发流量就会直接到达数据库。

什么是缓存雪崩？

如果在某一时刻缓存集中失效，或者缓存系统出现故障，所有的并发流量就会直接到达数据库。数据存储层的调用量就会暴增，用不了多长时间，数据库就会被大流量压垮，这种级联式的服务故障，就叫作缓存雪崩。

我们可以用下图来表示缓存雪崩的现象。



造成缓存雪崩的主要原因就是缓存集中失效，或者缓存服务发生故障，瞬间的大并发流量压垮了数据库。

如何解决缓存雪崩问题？

解决缓存雪崩问题最常用的一种方案就是保证Redis的高可用，将Redis缓存部署成高可用集群（必要时候做成异地多活），可以有效的防止缓存雪崩问题的发生。

为了缓解大并发流量，我们也可以使用限流降级的方式防止缓存雪崩。例如，在缓存失效后，通过加锁或者使用队列来控制读数据库写缓存的线程数量。具体点就是设置某些Key只允许一个线程查询数据和写缓存，其他线程等待。则能够有效的缓解大并发流量对数据库打来的巨大冲击。

另外，我们也可以通过数据预热的方式将可能大量访问的数据加载到缓存，在即将发生大并发访问的时候，提前手动触发加载不同的数据到缓存中，并为数据设置不同的过期时间，让缓存失效的时间点尽量均匀，不至于在同一时刻全部失效。

面试官：Java中提供了synchronized，为什么还要提供Lock呢？

写在前面

> 在Java中提供了synchronized关键字来保证只有一个线程能够访问同步代码块。既然已经提供了synchronized关键字，那为何在Java的SDK包中，还会提供Lock接口呢？这是不是重复造轮子，多此一举呢？今天，我们就一起来探讨下这个问题。

再造轮子？

既然JVM中提供了synchronized关键字来保证只有一个线程能够访问同步代码块，为何还要提供Lock接口呢？这是在重复造轮子吗？Java的设计者们为何要这样做呢？让我们一起带着疑问往下看。

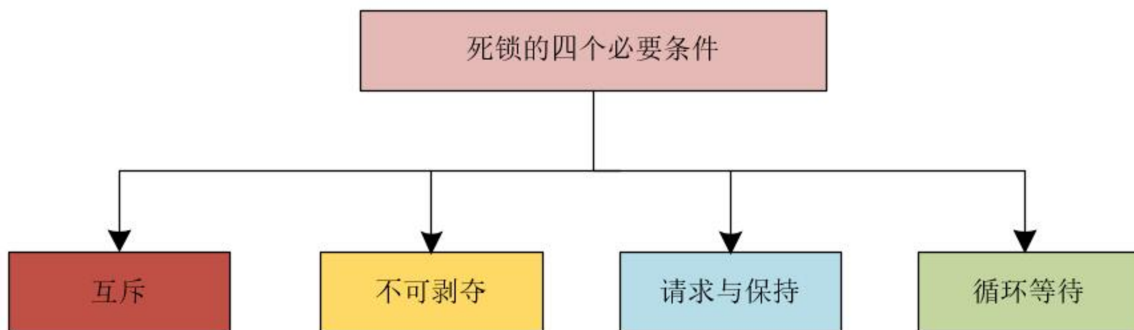
为何提供Lock接口？

很多小伙伴可能会听说过，在Java 1.5版本中，synchronized的性能不如Lock，但在Java 1.6版本之后，synchronized做了很多优化，性能提升了不少。那既然synchronized关键字的性能已经提升了，那为何还要使用Lock呢？

如果我们向更深层次思考的话，就不难想到了：我们使用synchronized加锁是无法主动释放锁的，这就会涉及到死锁的问题。

死锁问题

如果要发生死锁，则必须存在以下四个必要条件，四者缺一不可。



- **互斥条件**

在一段时间内某资源仅为一个线程所占有。此时若有其他线程请求该资源，则请求线程只能等待。

- **不可剥夺条件**

线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走，即只能由获得该资源的线程自己来释放（只能是主动释放)。

- **请求与保持条件**

线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时请求线程被阻塞，但对自己已获得的资源保持不放。

- **循环等待条件**

在发生死锁时必然存在一个进程等待队列{P1,P2,…,Pn},其中P1等待P2占有的资源，P2等待P3占有的资源，…，Pn等待P1占有的资源，形成一个进程等待环路，环路中每一个进程所占有的资源同时被另一个申请，也就是前一个进程占有后一个进程所深情地资源。

synchronized的局限性

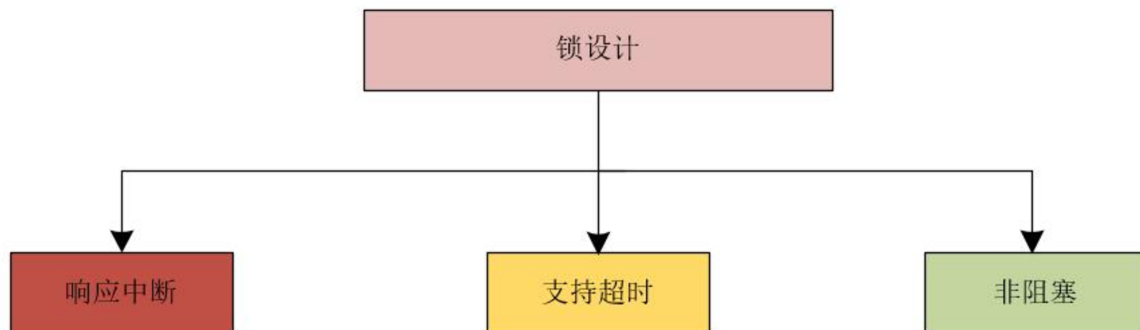
如果我们的程序使用synchronized关键字发生了死锁时，synchronized关键是是无法破坏“不可剥夺”这个死锁的条件的。这是因为synchronized申请资源的时候， 如果申请不到， 线程直接进入阻塞状态了， 而线程进入阻塞状态， 啥都干不了， 也释放不了线程已经占有的资源。

然而，在大部分场景下，我们都是希望“不可剥夺”这个条件能够被破坏。也就是说对于“不可剥夺”这个条件，占用部分资源的线程进一步申请其他资源时， 如果申请不到， 可以主动释放它占有的资源， 这样不可剥夺这个条件就破坏掉了。

如果我们自己重新设计锁来解决synchronized的问题，我们该如何设计呢?

解决问题

了解了synchronized的局限性之后，如果是让我们自己实现一把同步锁，我们该如何设计呢？也就是说，我们在设计锁的时候，要如何解决synchronized的局限性问题呢？这里，我觉得可以从三个方面来思考这个问题。



（1）能够响应中断。 synchronized的问题是， 持有锁A后， 如果尝试获取锁B失败， 那么线程就进入阻塞状态， 一旦发生死锁，就没有任何机会来唤醒阻塞的线程。 但如果阻塞状态的线程能够响应中断信号， 也就是说当我们给阻塞的线程发送中断信号的时候， 能够唤醒它， 那它就有机会释放曾经持有的锁A。 这样就破坏了不可剥夺条件了。

（2）支持超时。 如果线程在一段时间之内没有获取到锁， 不是进入阻塞状态， 而是返回一个错误， 那这个线程也有机会释放曾经持有的锁。 这样也能破坏不可剥夺条件。

（3）非阻塞地获取锁。 如果尝试获取锁失败， 并不进入阻塞状态， 而是直接返回， 那这个线程也有机会释放曾经持有的锁。 这样也能破坏不可剥夺条件。

体现在Lock接口上，就是Lock接口提供的三个方法，如下所示。

```
// 支持中断的API
void lockInterruptibly() throws InterruptedException;
// 支持超时的API
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
// 支持非阻塞获取锁的API
boolean tryLock();
```

- lockInterruptibly()

支持中断。

- tryLock()方k

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

- tryLock(long time, TimeUnit unit)方法

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

也就是说，对于死锁问题，Lock能够破坏不可剥夺的条件，例如，我们下面的程序代码就破坏了死锁的不可剥夺的条件。

```
public class TansferAccount{
    private Lock thisLock = new ReentrantLock();
    private Lock targetLock = new ReentrantLock();
    //账户的余额
    private Integer balance;
    //转账操作
    public void transfer(TansferAccount target, Integer transferMoney){
        boolean isThisLock = thisLock.tryLock();
        if(isThisLock){
            try{
                boolean isTargetLock = targetLock.tryLock();
                if(isTargetLock){
                    try{
                         if(this.balance >= transferMoney){
                            this.balance -= transferMoney;
                            target.balance += transferMoney;
                        }
                    }finally{
                        targetLock.unlock
                    }
                }
```

```
            }finally{
                thisLock.unlock();
            }
        }
    }
}
```

例外，Lock下面有一个ReentrantLock，而ReentrantLock支持公平锁和非公平锁。

在使用ReentrantLock的时候， ReentrantLock中有两个构造函数， 一个是无参构造函数， 一个是传入fair参数的构造函数。 fair参数代表的是锁的公平策略， 如果传入true就表示需要构造一个公平锁， 反之则表示要构造一个非公平锁。如下代码片段所示。

```
//无参构造函数： 默认非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}
//根据公平策略参数创建锁
public ReentrantLock(boolean fair){
    sync = fair ? new FairSync() : new NonfairSync();
}
```

锁的实现在本质上都对应着一个入口等待队列， 如果一个线程没有获得锁， 就会进入等待队列， 当有线程释放锁的时候， 就需要从等待队列中唤醒一个等待的线程。 如果是公平锁， 唤醒的策略就是谁等待的时间长， 就唤醒谁， 很公平； 如果是非公平锁， 则不提供这个公平保证， 有可能等待时间短的线程反而先被唤醒。 而Lock是支持公平锁的，synchronized不支持公平锁。

最后，值得注意的是，在使用Lock加锁时，一定要在finally{}代码块中释放锁，例如，下面的代码片段所示。

```
try{
    lock.lock();
}finally{
    lock.unlock();
}
```

注：其他synchronized和Lock的详细说明，小伙伴们自行查阅即可。

面试官：说说缓存最关心的问题是什么？有哪些类型？回收策略和算法？

写在前面

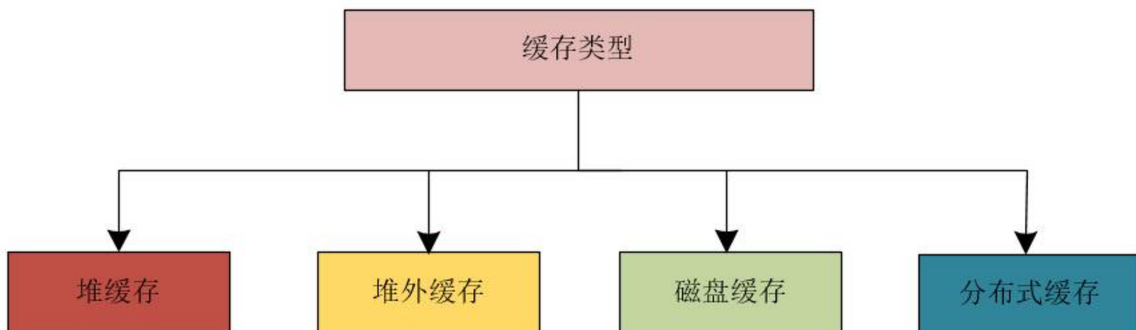
> 往往开始做一个项目时，不会过多的考虑性能问题，以快速迭代功能为主。后续随着业务的快速发展，系统运行的性能越来越慢，此时，就需要对系统进行相应的优化，而效果最显著的就是给系统加上缓存。那么，问题来了，当你为系统加上缓存时，有没有考虑过使用缓存需要注意哪些事项呢？

缓存命中率

缓存命中率是从缓存中读取数据的次数与总读取次数的比率，命中率越高越好。**缓存命中率=从缓存中读取次数 / (总读取次数 (从缓存中读取次数 + 从慢速设备上读取次数))**。这是一个非常重要的监控指标，如果做缓存，则应通过监控这个指标来看缓存是否工作良好。

缓存类型

缓存类型总体上来看，可以分为：堆缓存、堆外缓存、磁盘缓存和分布式缓存。



堆内存

使用Java堆内存来存储对象。使用堆缓存的好处是没有序列化/反序列化，是最快的缓存。缺点也很明显，当缓存的数据量很大时，GC（垃圾回收）暂停时间会变长，存储容量受限于堆空间大小。一般通过软引用/弱引用来存储缓存对象。即当堆内存不足时，可以强制回收这部分内存释放堆内存空间。一般使用堆缓存存储较热的数据。可以使用Guava Cache、Ehcache 3.x、 MapDB实现。

堆外内存

即缓存数据存储在堆外内存，可以减少GC暂停时间（堆对象转移到堆外，GC扫描和移动的对象变少了），可以支持更多的缓存空间（只受机器内存大小限制，不受堆空间的影响）。但是，读取数据时需要序列化/反序列化。因此，会比堆缓存慢很多。可以使用Ehcache 3.x、 MapDB实现。

磁盘缓存

即缓存数据存储在磁盘上，在JVM重启时数据还存在，而堆/堆外缓存数据会丢失，需要重新加载。可以使用Ehcache 3.x、MapDB实现。

分布式缓存

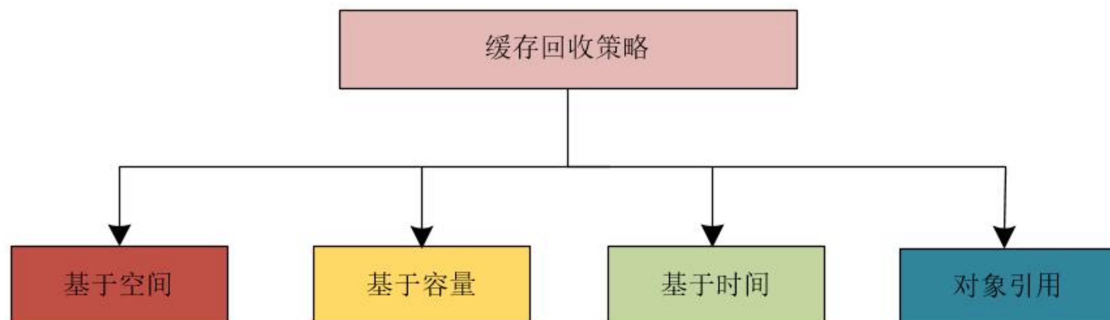
分布式缓存可以使用ehcache-clustered(配合Terracotta server)实现Java进程间分布式缓存。也可以使用Memcached、Redis实现。

使用分布式缓存时，有两种模式如下：

- 单机模式：存储最热的数据到堆缓存，相对热的数据到堆外缓存，不热的数据到磁盘缓存。
- 集群模式：存储最热的数据到堆缓存，相对热的数据到对外缓存，全量数据到分布式缓存。

缓存回收策略

缓存的回收策略总体上来说包含：基于空间的回收策略、基于容量（空间）的回收策略、基于时间的回收策略和基于对象引用的回收策略。



基于空间

基于空间指缓存设置了存储空间，如设置为10MB，当达到存储空间上限时，按照一定的策略移除数据。

基于容量

基于容量指缓存设置了最大大小，当缓存的条目超过最大大小时，按照一定的策略移除旧数据。

基于时间

TTL(Time To Live):存活期，即缓存数据从创建开始直到到期的一个时间段（不管在这个时间段内有没有被访问，缓存数据都将过期）。
TTI(Time To Idle)：空闲期，即缓存数据多久没被访问后移除缓存的时间。

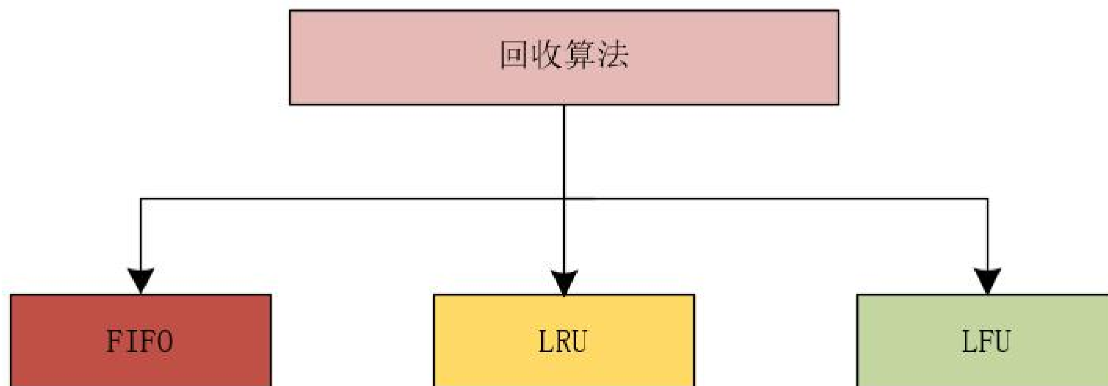
基于对象引用

软引用：如果一个对象是软引用，则当JVM堆内存不足时，垃圾回收器可以回收这些对象。软引用适合用来做缓存，从而当JVM堆内存不足时，可以回收这些对象腾出一些空间供强引用对象使用，从而避免OOM。
弱引用：当垃圾回收器回收内存时，如果发现弱引用，则将它立即回收。相对于软引用，弱引用有更短的生命周期。

注意：只有在没有其他强引用对象引用弱引用/软引用对象时，垃圾回收时才回收该引用。即如果有一个对象（不是弱引用/软引用对象）引用了弱引用/软引用对象，那么垃圾回收时不会回收该弱引用/软引用对象。

回收算法

使用基于空间和基于容量的缓存会使用一定的策略移除旧数据，通常包含：FIFO算法、LRU算法和LFU算法。



- FIFO(First In First Out)：先进先出算法，即先放入缓存的先被移除。
- LRU(Least Recently Used)：最近最少使用算法，时间时间距离现在最久的那个被移除。
- LFU(Least Frequently Used)：最不常用算法，一定时间段内使用次数（频率）最少的那个被移除。

实际应用中基于LRU的缓存居多。

好了，今天就聊到这儿吧！别忘了点个赞，给个在看和转发，让更多的人看到，一起学习，一起进步！！

面试官：性能优化有哪些衡量指标？需要注意什么？

写在前面

> 最近，很多小伙伴都在说，我没做过性能优化的工作，在公司只是做些CRUD的工作，接触不到性能优化相关的工作。现在出去找工作面试的时候，面试官总是问些很刁钻的问题来为难我，很多我都不会啊！那怎么办呢？那我就专门写一些与高并发系统相关的面试容易问到的问题吧。今天，我们就来说说在高并发场景下做性能优化有哪些衡量标准，以及做优化时需要注意哪些问题。

面试场景

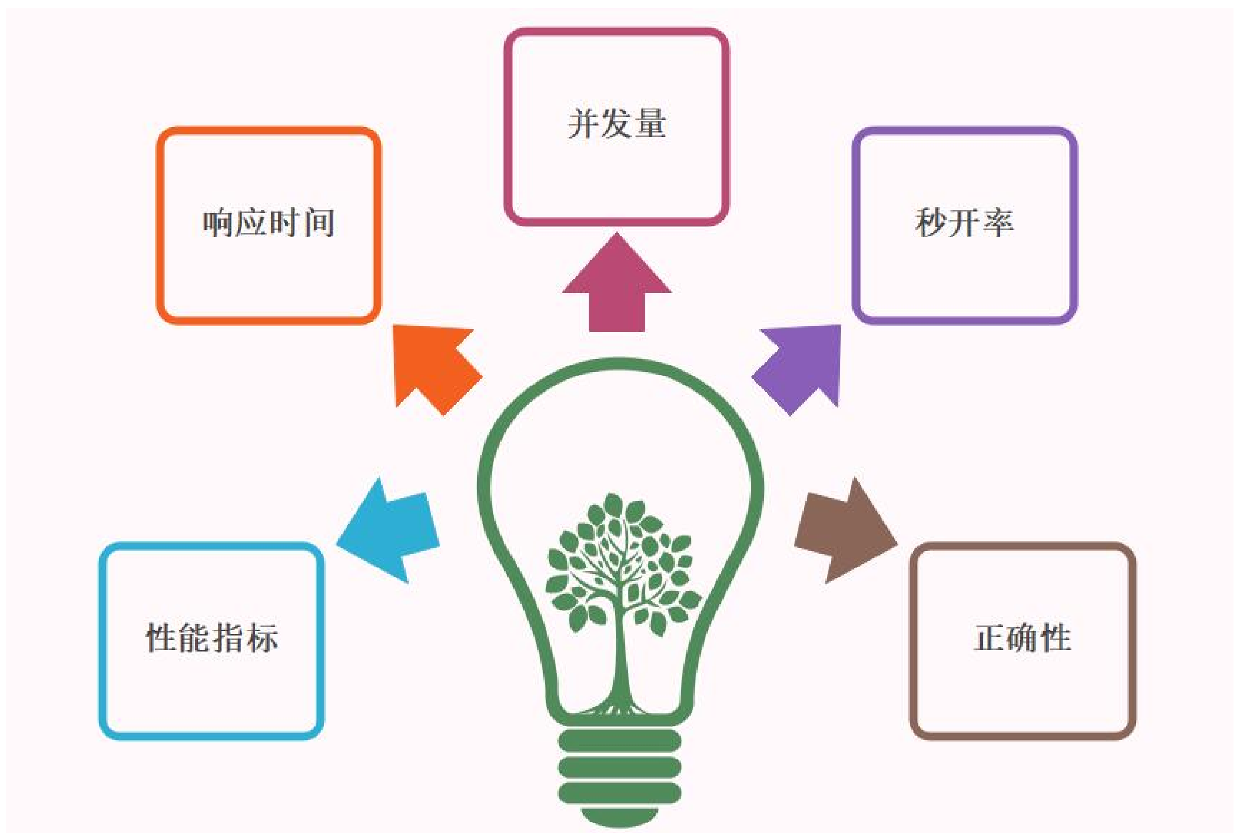
面试官：平时工作中有没有做过一些性能优化相关的工作呢？

首先，我们来分析下面试官的这个问题。其实，以我本人招聘面试的经验来说，如果面试官问出了这样的一个问题。本质上不只是想让面试者简单的回答：做过或者没做过。而是想通过这个简单的问题来考察下面试者的思考能力和对于问题的理解能力。面试官本质上是想让面试者通过这个问题，讲述一下自己做性能优化相关工作的经验、以及对于性能优化工作的一些理论的理解，比如就包括：性能优化的衡量指标，期间需要注意的问题等等。

如果面试者在面试过程中，不能充分理解面试官的意图，回答问题时，像挤牙膏一样，挤点出点，那么，大多数情况下，面试官就会认为这个人没啥性能优化的经验。此时，面试者就会在面试官心理的印象大打折扣，面试结果就有非常大的概率凉凉了。

衡量指标

对于性能优化来说，衡量的指标有很多，大体上可以分为：性能指标、响应时间、并发量、秒开率和正确性等。我们可以使用下图来表示这些衡量指标。



接下来，我们就分别说明下这些衡量指标。

性能指标

性能指标又可以包含：吞吐量和响应速度。我们平时所说的QPS、TPS和HPS等，就可以归结为吞吐量。有很多小伙伴可能对于QPS、TPS和HPS等不太了解，我们先来说下这几个字母的含义。

- QPS代表的是每秒的查询数量。
- TPS代表的是每秒事务的数量。
- HPS代表的是每秒的HTTP请求数量。

这些都是与吞吐量相关的衡量指标。

平时我们在做优化工作的时候，首先要明确需要优化的事项。比如：我们做的优化工作是要提高系统的吞吐量？还是要提升系统的响应速度呢？举一个具体点的例子：比如我们的程序中存在一些数据库或者缓存的批量操作，虽然在数据的读取上，响应速度下降了，但是我们优化的目标就是吞吐量，只要我们优化后系统的整体吞吐量明显上升了，那这也是提升了程序的性能。

所以说，优化性能不只是提升系统的响应速度。

这里，优化性能也并不是一味的优化吞吐量和优化响应速度，而是在吞吐量和响应速度之间找到一个平衡点，使用有限的服务器资源来更好的提升用户体验。

响应时间

对于响应时间来说，有两个非常重要的衡量指标。那就是：**平均响应时间和百分位数。**

（1）平均响应时间

通常，平均响应时间体现的是服务接口的平均处理能力。计算方式就是把所有的请求所耗费的时间加起来，然后除以请求的次数。举个简单的例子：比如：我们向一个网站发送了5次请求，每次请求所耗费的时间分别为：1ms，2ms，1ms，3ms，2ms，那么，平均响应时间就是（1+2+1+3+2）/ 5 = 1.8ms，所以，平均响应时间就是1.8ms。

平均响应时间这个指标存在一个问题：如果在短时间内请求变得很慢，但很快过去了，此时使用平均响应时间就无法很好的体现出性能的波动问题。

（2）百分位数

百分位数就是我们在优化的时候，圈定一个时间范围，把每次请求的耗时加入一个列表中，然后按照从小到大的顺序将这些时间进行排序。这样，我们取出特定百分位的耗时，这个数字就是 TP 值。

TP值表示的含义就是：超过 N% 的请求都在 X 时间内返回。比如 TP90 = 50ms，意思是超过 90th 的请求，都在 50ms 内返回。

百分位数这个指标也是很重要的，它反映的是应用接口的整体响应情况。

我们一般会将百分位数分为 TP50、TP90、TP95、TP99、TP99.9 等多个段，对高百分位的值要求越高，对系统响应能力的稳定性要求越高。

并发量

并发量指的是系统能够同时处理的请求数量，反映的是系统的负载能力。

我们在对高并发系统进行优化的时候，往往也会在并发量上进行调优，调优方式也是多种多样的，目的就是提高系统同时处理请求的能力。

总体来说，并发量这个指标理解起来还是比较简单的，我就不做过多的描述了。

秒开率

秒开率主要针对的是前端网页或者移动端APP来说的，如果一个前端网页或者APP能够在1秒内很平滑的打开，尤其是首页的加载。此时，用户就会感到前端网页或者APP使用起来很顺畅，如果超过3秒甚至更长的时间，用户就有可能会直接退出前端网页或者APP不再使用。

所以，在高并发场景下优化程序，不只要对后端程序进行优化，对于前端和APP也是要进行优化的。

正确性

正确性说的是无论我们以何种方式，何种手段对应用进行优化，优化后的交互数据结果必须是正确的。不能出现优化前性能比较低，数据正确，而优化后性能比较高，反而数据不正确的现象。

优化需要注意的问题

- 除非必要,一开始不要优化(尤其是开发阶段)
- 有些优化准则已经过时,需要考虑当下的软硬件环境(不要墨守成规)
- 不要过分强调某些系统级指标,如cache 命中率,而应该聚焦性能瓶颈点
- 不盲从，测试、找到系统的性能瓶颈，再确定优化手段
- 注意权衡优化的成本和收益（有些优化可能需要现有架构做出调整、增加开发/运维成本）
- 优化的目标是用户体验、降低硬件成本（降低集群规模、不依赖单机高性能）
- 测试环境的优化手段未必对生产环境有效（优化需要针对真实情况）

面试官问我如何使用Nginx实现限流，我如此回答轻松拿到了Offer！

写在前面

> 最近，有不少读者说看了我的文章后，学到了很多知识，其实我本人听到后是非常开心的，自己写的东西能够为大家带来帮助，确实是一件值得高兴的事情。最近，也有不少小伙伴，看了我的文章后，顺利拿到了大厂Offer，也有不少小伙伴一直在刷我的文章，提升自己的内功，最终成为自己公司的核心业务开发人员。在此，冰河确实为你们高兴，希望小伙伴们能够一如既往的学习，保持一颗持续学习的心态，在技术的道路上越走越远。
>
> 今天写些什么呢？想来想去，写一篇关于高并发实战的文章吧，对，就写一下如何使用Nginx实现限流的文章吧。小伙伴们想看什么文章，可以在微信上给我留言，或者直接在公众号留言。

限流措施

如果看过我写的《【高并发】高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！》一文的话，相信小伙伴们都会记得我说过的：**网上很多的文章和帖子中在介绍秒杀系统时，说是在下单时使用异步削峰来进行一些限流操作，那都是在扯淡！因为下单操作在整个秒杀系统的流程中属于比较靠后的操作了，限流操作一定要前置处理，在秒杀业务后面的流程中做限流操作是没啥卵用的。**

Nginx作为一款高性能的Web代理和负载均衡服务器，往往会部署在一些互联网应用比较前置的位置。此时，我们就可以在Nginx上进行设置，对访问的IP地址和并发数进行相应的限制。

Nginx官方的限流模块

Nginx官方版本限制IP的连接和并发分别有两个模块：

- limit_req_zone 用来限制单位时间内的请求数，即速率限制,采用的漏桶算法 "leaky bucket"。
- limit_req_conn 用来限制同一时间连接数，即并发限制。

limit_req_zone 参数配置

limit_req_zone参数说明

```
Syntax: limit_req zone=name [burst=number] [nodelay];
Default:    —
Context:    http, server, location
```

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

- 第一个参数：$binary_remote_addr 表示通过remote_addr这个标识来做限制，“binary_”的目的是缩写内存占用量，是限制同一客户端ip地址。
- 第二个参数：zone=one:10m表示生成一个大小为10M，名字为one的内存区域，用来存储访问的频次信息。
- 第三个参数：rate=1r/s表示允许相同标识的客户端的访问频次，这里限制的是每秒1次，还可以有比如30r/m的。

```
limit_req zone=one burst=5 nodelay;
```

- 第一个参数：zone=one 设置使用哪个配置区域来做限制，与上面limit_req_zone 里的name对应。
- 第二个参数：burst=5，重点说明一下这个配置，burst爆发的意思，这个配置的意思是设置一个大小为5的缓冲区当有大量请求（爆发）过来时，超过了访问频次限制的请求可以先放到这个缓冲区内。
- 第三个参数：nodelay，如果设置，超过访问频次而且缓冲区也满了的时候就会直接返回503，如果没有设置，则所有请求会等待排队。

limit_req_zone示例

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
    server {
        location /search/ {
            limit_req zone=one burst=5 nodelay;
        }
}
```

下面配置可以限制特定UA（比如搜索引擎）的访问：

```
limit_req_zone  $anti_spider  zone=one:10m   rate=10r/s;
limit_req zone=one burst=100 nodelay;
if ($http_user_agent ~* "googlebot|bingbot|Feedfetcher-Google") {
    set $anti_spider $http_user_agent;
}
```

其他参数

```
Syntax: limit_req_log_level info | notice | warn | error;
Default:
limit_req_log_level error;
Context:    http, server, location
```

当服务器由于limit被限速或缓存时，配置写入日志。延迟的记录比拒绝的记录低一个级别。例子：`limit_req_log_level notice` 延迟的的基本是info。

```
Syntax: limit_req_status code;
Default:
limit_req_status 503;
Context:    http, server, location
```

设置拒绝请求的返回值。值只能设置 400 到 599 之间。

ngx_http_limit_conn_module 参数配置

ngx_http_limit_conn_module 参数说明

这个模块用来限制单个IP的请求数。并非所有的连接都被计数。只有在服务器处理了请求并且已经读取了整个请求头时，连接才被计数。

```
Syntax: limit_conn zone number;
Default:    —
Context:    http, server, location
```

```
limit_conn_zone $binary_remote_addr zone=addr:10m;

server {
    location /download/ {
        limit_conn addr 1;
    }
```

一次只允许每个IP地址一个连接。

```
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_conn_zone $server_name zone=perserver:10m;

server {
    ...
    limit_conn perip 10;
    limit_conn perserver 100;
}
```

可以配置多个limit_conn指令。例如，以上配置将限制每个客户端IP连接到服务器的数量，同时限制连接到虚拟服务器的总数。

```
Syntax: limit_conn_zone key zone=name:size;
Default:    —
Context:    http
```

```
limit_conn_zone $binary_remote_addr zone=addr:10m;
```

在这里，客户端IP地址作为关键。请注意，不是$remote_addr$，而是使用 binary_remote_addr变量。
$remote_addr$变量的大小可以从7到15个字节不等。存储的状态在32位平台上占用32或64字节的内存，在64位平台上总是占用64字节。对于$IPv4$地址，
binary_remote_addr变量的大小始终为4个字节，对于IPv6地址则为16个字节。存储状态在32位平台上始终占用32或64个字节，在64位平台上占用64个字节。一个兆字节的区域可以保持大约32000个32字节的状态或大约16000个64字节的状态。如果区域存储耗尽，服务器会将错误返回给所有其他请求。

```
Syntax: limit_conn_log_level info | notice | warn | error;
Default:
limit_conn_log_level error;
Context:    http, server, location
```

当服务器限制连接数时，设置所需的日志记录级别。

```
Syntax: limit_conn_status code;
Default:
limit_conn_status 503;
Context:    http, server, location
```

设置拒绝请求的返回值。

Nginx限流实战

限制访问速率

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=2r/s;
server {
    location / {
        limit_req zone=mylimit;
    }
}
```

上述规则限制了每个IP访问的速度为2r/s，并将该规则作用于根目录。如果单个IP在非常短的时间内并发发送多个请求，结果会怎样呢?

S...	Status	Connect Time(ms)	Sample Time(ms)	Start Time	Thread Name	Bytes	Sent Bytes	Latency	Label
1	✔	1	3	10:58:37.060	add burst 1-1	434	126	3	HTTP Req...
2	✘	1	2	10:58:37.230	add burst 1-2	324	126	2	HTTP Req...
3	✘	1	2	10:58:37.396	add burst 1-3	324	126	2	HTTP Req...
4	✘	1	1	10:58:37.565	add burst 1-4	324	126	1	HTTP Req...
5	✘	1	3	10:58:37.730	add burst 1-5	324	126	3	HTTP Req...
6	✘	1	2	10:58:37.896	add burst 1-6	324	126	2	HTTP Req...

https://blog.csdn.net/l1028386804

我们使用单个IP在10ms内发并发送了6个请求，只有1个成功，剩下的5个都被拒绝。我们设置的速度是2r/s，为什么只有1个成功呢，是不是Nginx限制错了？当然不是，是因为Nginx的限流统计是基于毫秒的，我们设置的速度是2r/s，转换一下就是500ms内单个IP只允许通过1个请求，从501ms开始才允许通过第二个请求。

burst缓存处理

我们看到，我们短时间内发送了大量请求，Nginx按照毫秒级精度统计，超出限制的请求直接拒绝。这在实际场景中未免过于苛刻，真实网络环境中请求到来不是匀速的，很可能有请求"突发"的情况，也就是"一股子一股子"的。Nginx考虑到了这种情况，可以通过burst关键字开启对突发请求的缓存处理，而不是直接拒绝。

来看我们的配置：

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=2r/s;
server {
    location / {
        limit_req zone=mylimit burst=4;
    }
}
```

我们加入了burst=4，意思是每个key(此处是每个IP)最多允许4个突发请求的到来。如果单个IP在10ms内发送6个请求，结果会怎样呢?

S...	Status	Connect Time(ms)	Sample Time(ms)	Start Time	Thread Name	Bytes	Sent Bytes	Latency	Label
1	✔	9	15	10:55:45.996	add burst 1-1	434	126	15	HTTP Req...
2	✘	1	2	10:55:46.845	add burst 1-6	324	126	2	HTTP Req...
3	✔	0	832	10:55:46.179	add burst 1-2	434	126	832	HTTP Req...
4	✔	0	1667	10:55:46.345	add burst 1-3	434	126	1667	HTTP Req...
5	✔	1	2500	10:55:46.512	add burst 1-4	434	126	2500	HTTP Req...
6	✔	1	3334	10:55:46.678	add burst 1-5	434	126	3334	HTTP Req...

https://blog.csdn.net/l1028386804

相比实例一成功数增加了4个，这个我们设置的burst数目是一致的。具体处理流程是：1个请求被立即处理，4个请求被放到burst队列里，另外一个请求被拒绝。通过burst参数，我们使得Nginx限流具备了缓存处理突发流量的能力。

但是请注意：burst的作用是让多余的请求可以先放到队列里，慢慢处理。如果不加nodelay参数，队列里的请求不会立即处理，而是按照rate设置的速度，以毫秒级精确的速度慢慢处理。

nodelay降低排队时间

在使用burst缓存处理中，我们看到，通过设置burst参数，我们可以允许Nginx缓存处理一定程度的突发，多余的请求可以先放到队列里，慢慢处理，这起到了平滑流量的作用。但是如果队列设置的比较大，请求排队的时间就会比较长，用户角度看来就是RT变长了，这对用户很不友好。有什么解决办法呢？nodelay参数允许请求在排队的时候就立即被处理，也就是说只要请求能够进入burst队列，就会立即被后台worker处理，请注意，这意味着burst设置了nodelay时，系统瞬间的QPS可能会超过rate设置的阈值。nodelay参数要跟burst一起使用才有作用。

延续burst缓存处理的配置，我们加入nodelay选项：

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=2r/s;
server {
    location / {
        limit_req zone=mylimit burst=4 nodelay;
    }
}
```

单个IP 10ms内并发发送6个请求，结果如下：

Start Time	Status	Connect Time(ms)	Thread Name	Sample Time(ms)	Bytes	Sent Bytes	Latency	Label
10:52:39.798	✓	2	add burst 1-1	3	434	126	3	HTTP Re...
10:52:39.969	✓	1	add burst 1-2	834	434	126	833	HTTP Re...
10:52:40.134	✓	1	add burst 1-3	1668	434	126	1668	HTTP Re...
10:52:40.302	✓	1	add burst 1-4	2502	434	126	2501	HTTP Re...
10:52:40.469	✓	1	add burst 1-5	3334	434	126	3334	HTTP Re...
10:52:40.636	✗	2	add burst 1-6	3	324	126	3	HTTP Re...
10:51:51.757	✓	1	add nodelay 1-1	2	434	126	2	HTTP Re...
10:51:51.926	✓	0	add nodelay 1-2	1	434	126	1	HTTP Re...
10:51:52.094	✓	1	add nodelay 1-3	2	434	126	2	HTTP Re...
10:51:52.261	✓	1	add nodelay 1-4	3	434	126	3	HTTP Re...
10:51:52.427	✓	1	add nodelay 1-5	2	434	126	2	HTTP Re...
10:51:52.594	✗	0	add nodelay 1-6	1	324	126	1	HTTP Re...

跟burst缓存处理相比，请求成功率没变化，但是总体耗时变短了。这怎么解释呢？在burst缓存处理中，有4个请求被放到burst队列当中，工作进程每隔500ms(rate=2r/s)取一个请求进行处理，最后一个请求要排队2s才会被处理；这里，请求放入队列跟burst缓存处理是一样的，但不同的是，队列中的请求同时具有了被处理的资格，所以这里的5个请求可以说是同时开始被处理的，花费时间自然变短了。

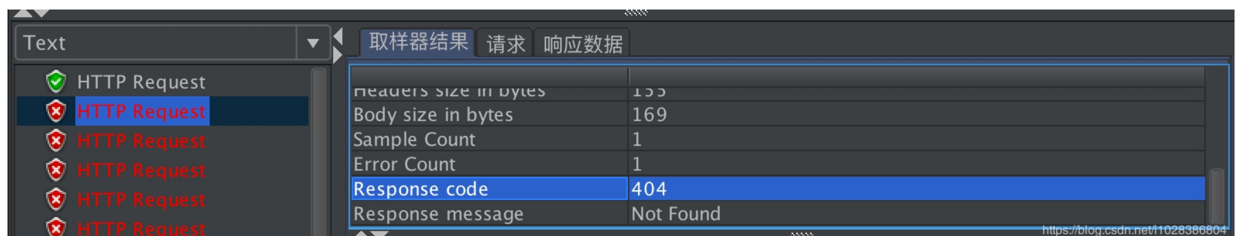
但是请注意，虽然设置burst和nodelay能够降低突发请求的处理时间，但是长期来看并不会提高吞吐量的上限，长期吞吐量的上限是由rate决定的，因为nodelay只能保证burst的请求被立即处理，但Nginx会限制队列元素释放的速度，就像是限制了令牌桶中令牌产生的速度。

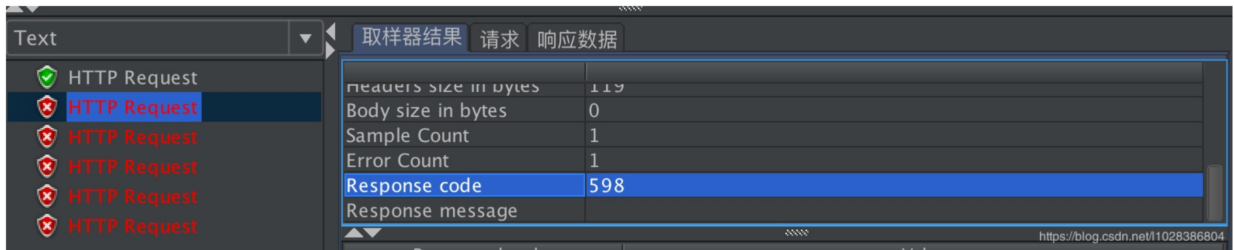
看到这里你可能会问，加入了nodelay参数之后的限速算法，到底算是哪一个"桶"，是漏桶算法还是令牌桶算法？当然还算是漏桶算法。考虑一种情况，令牌桶算法的token为耗尽时会怎么做呢？由于它有一个请求队列，所以会把接下来的请求缓存下来，缓存多少受限于队列大小。但此时缓存这些请求还有意义吗？如果server已经过载，缓存队列越来越长，RT越来越高，即使过了很久请求被处理了，对用户来说也没什么价值了。所以当token不够用时，最明智的做法就是直接拒绝用户的请求，这就成了漏桶算法。

自定义返回值

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=2r/s;
server {
    location / {
        limit_req zone=mylimit burst=4 nodelay;
        limit_req_status 598;
    }
}
```

默认情况下 没有配置 status 返回值的状态：





好了，咱们今天就聊到这儿吧！别忘了给个在看和转发，让更多的人看到，一起学习一起进步！！

如何设计一个支撑高并发大流量的系统？

写在前面

> 最近不少小伙伴们都在问我：高并发专题我学了不少文章了，但是如何设计一个高并发的系统我还是一脸懵逼！这个问题怎么解决呢？其实，相信不只是问我的这些小伙伴有这个困惑，就连工作（入坑）了好几年的开发人员也都有这样的困惑：我学习了很多的高并发课程，也看了不少的高大上的文章，可就是不知道怎么去设计一个支撑高并发大流量的系统。针对小伙伴们的疑惑，这里，我就把一些设计高并发大流量的常规思路分享给大家，不一定完全正确，设计高并发大流量系统本来就是一个仁者见仁、智者见智的事情，只要是符合自身业务场景的架构思路，都是好的架构思路，架构本身来说就是没有一个完全正确的架构，而是尽量符合当时自身的业务场景，并且能够良好的支撑业务的负载。

高并发架构相关概念

什么是并发？

并发是指并发的访问，也就是某个时间点，有多少个访问同时到来；

通常如果一个系统的日PV在千万以上，有可能是一个高并发的系统，这里需要注意的是：只是有可能是一个高并发的系统，不一定是一个高并发的系统。

并发数和QPS是不同的概念，一般说QPS会说多少并发用户下QPS，当QPS相同时，并发用户数越大，网站并发处理能力越好。当并发用户数过大时，会造成进程（线程）频繁切换，反正真正用于处理请求的时间变少，每秒能够处理的请求数反而变少，同时用户的请求等待时间也会变大。 找到最佳线程数能够让web系统更稳定，效率更高。

并发数 = QPS*平均响应时间

高并发具体关心什么？

QPS： 每秒请求或查询的数量，在互联网领域，指每秒响应请求数；

吞吐量： 单位时间内处理的请求量（通常由QPS与并发数决定）；

响应时间： 从请求发出到收到响应花费的时间，例如一个系统处理一个HTTP请求需要100ms，这个100ms就是系统的响应时间；

PV： 综合浏览量，即页面浏览量或者点击量，一个访客在24小时内访问的页面数量；

UV： 独立访客 ，即一定时间范围内相同访客多次访问网站，只计算为一个独立的访客；

带宽： 计算带宽大小需要关注两个指标，**峰值流量和页面的平均大小**；

日网站带宽可以使用下面的公式来粗略计算：

```
日网站带宽=pv/统计时间（换算到秒）*平均页面大小（单位kB）*8
```

峰值一般是平均值的倍数；

QPS不等于并发连接数，QPS是每秒HTTP请求数量，并发连接数是系统同时处理的请求数量；

```
峰值每秒请求数（QPS） = (总PV数 * 80%) /（6小时秒数 * 20%）
```

压力测试： 测试能承受的最大并发，测试最大承受的QPS值。

测试工具（ab）： 目标是URL，可以创建多个访问线程对同一个URL进行访问（Nginx）；

ab的使用： 模拟并发请求100次（100个人），总共请求5000次（每个人请求5000次）

```
ab -c 100 -n 5000 待测试网站（内存和网络不超过最高限度的75%）
```

QPS达到50： 一般的服务器就可以应付；

QPS达到100： 假设关系型数据库的每次请求在0.01秒完成（理想），假设单页面只有一个SQL查询，那么100QPS意味着1秒中完成100次请求，但此时我们不能保证数据库查询能完成100次；

方案：数据库缓存层、数据库的负载均衡；

QPS达到800： 假设我们使用 百兆宽带，意味着网站出口的实际带宽是8M左右，假设每个页面是有10k，在这个并发的条件下，百兆带宽已经被吃完；

方案：CDN加速、负载均衡

QPS达到1000： 假设使用Redis缓存数据库查询数据，每个页面对Redis请求远大于直接对DB的请求；
Redis的悲观并发数在5W左右，但有可能之前内网带宽已经被吃光，表现出不稳定；

方案：静态HTML缓存

QPS达到2000： 文件系统访问锁都成为了灾难；

方案：做业务分离，分布式存储；

高并发解决方案案例

流量优化： 防盗链处理（把一些恶意的请求拒之门外）

前端优化： 减少HTTP请求、添加异步请求、启用浏览器的缓存和文件压缩、CDN加速、建立独立的图片服务器；

服务端优化： 页面静态化处理、并发处理、队列处理；

数据库优化： 数据库的缓存、分库分表、分区操作、读写分离、负载均衡

Web服务器优化： 负载均衡

高并发下的经验公式

通过QPS和PV计算部署服务器的台数

单台服务器每天PV计算

```
公式1：每天总PV = QPS * 3600 * 6
公式2：每天总PV = QPS * 3600 * 8
```

服务器计算

```
服务器数量 =   ceil( 每天总PV / 单台服务器每天总PV )
```

峰值QPS和机器计算公式

原理： 每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间

公式： (总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

机器： 峰值时间每秒QPS / 单台机器的QPS = 需要的机器。

关于乐观锁和悲观锁，蚂蚁金服面试官问了我这几个问题！！

写在前面

> 最近，一名读者去蚂蚁金服面试，面试官问了他关于乐观锁和悲观锁的问题，幸亏他看了我的【高并发专题】文章，结果是替这名读者高兴！现就部分面试题目总结成文，供小伙伴们参考。
>
> 小伙伴们可以关注 冰河技术 微信公众号来学习【高并发专题】，学习超硬核知识技能，跳槽大厂，升级加薪，指日可待！

何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以
场景而定说一种人好于另外一种人。

悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。 Java 中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。

乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。 乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

乐观锁的实现方式

乐观锁常见的两种实现方式：乐观锁一般会使用版本号机制或 CAS 算法实现。

版本号机制

一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时， version 值会加一。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子： 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ； 而当前帐户余额字段（balance ）为 $100 。

1. 操作员 A 此时将其读出（version=1 ），并从其帐户余额中扣除 50（100-$50 ）。
2. 在操作员 A 操作的过程中，操作员 B 也读入此用户信息（version=1 ），并从其帐户余额中扣除 20（100-$20 ）。
3. 操作员 A 完成了修改工作，将数据版本号加一（version=2 ），连同帐户扣除后余额（balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录version 更新为 2 。
4. 操作员 B 完成了操作，也将版本号加一（version=2 ）试图向数据库提交数据（balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 " 提交版本必须大于记录当前版本才能执行更新 " 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

CAS 算法

即 compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blockingSynchronization）。 CAS 算法涉及到三个操作数：

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B
  当且仅当 V 的值等于 A 时， CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点

ABA 问题是乐观锁一个常见的问题。

ABA 问题

如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。这个问题被称为 CAS 操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大

自旋 CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU 带来非常大的执行开销。 如果 JVM 能支持处理器提供的 pause 指令那么效率会有一定的提升， pause 指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline） ,使 CPU 不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起 CPU 流水线被清空（CPU pipeline flush），从而提高 CPU 的执行效率。

只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5 开始，提供了 AtomicReference 类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用 AtomicReference 类把多个共享变量合并成一个共享变量来操作。

CAS 与 synchronized 的使用场景

简单的来说 CAS 适用于写比较少的情况下（多读场景，冲突一般较少），synchronized 适用于写比较多的情况下（多写场景，冲突一般较多）

- 对于资源竞争较少（线程冲突较轻）的情况，使用 synchronized 同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu 资源；而 CAS 基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
- 对于资源竞争严重（线程冲突严重）的情况， CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized

补充： Java 并发编程这个领域中 synchronized 关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在JavaSE 1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。 synchronized 的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞， 竞争切换后继续竞争锁， 稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于CAS。

关于线程池，蚂蚁金服面试官问了我这些内容！！

写在前面

> 最近，一名读者去蚂蚁金服面试，面试官问了他关于乐观锁和悲观锁的问题，幸亏他看了我的【高并发专题】文章，结果是替这名读者高兴！现就部分面试题目总结成文，供小伙伴们参考。
>
> 小伙伴们可以关注 冰河技术 微信公众号来学习【高并发专题】，学习超硬核知识技能，跳槽大厂，升级加薪，指日可待！

面试汇总

Java中的线程池是如何实现的?

在Java中，所谓的线程池中的“线程”，其实是被抽象为了一个静态内部类Worker，它基于AQS实现，存放在线程池的HashSetworkers成员变量中；而需要执行的任务则存放在成员变量workQueue（BlockingQueueworkQueue）中。这样，整个线程池实现的基本思想就是、从workQueue中不断取出需要执行的任务，放在Workers中进行处理。

创建线程池的几个核心构造参数?

Java中的线程池的创建其实非常灵活，我们可以通过配置不同的参数，创建出行为不同的线程池，这几个参数包括：

- corePoolSize：线程池的核心线程数。
- maximumPoolSize：线程池允许的最大线程数。
- keepAliveTime：超过核心线程数时闲置线程的存活时间。
- workQueue：任务执行前保存任务的队列，保存由execute方法提交的Runnable任务。

线程池中的线程是怎么创建的？是一开始就随着线程池的启动创建好的吗?

显然不是的。线程池默认初始化后不启动Worker，等待有请求时才启动。每当我们调用execute()方法添加一个任务时，线程池会做如下判断：

如果正在运行的线程数量小于corePoolSize，那么马上创建线程运行这个任务；

如果正在运行的线程数量大于或等于corePoolSize，那么将这个任务放入队列；

如果这时候队列满了，而且正在运行的线程数量小于maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；

如果队列满了，而且正在运行的线程数量大于或等于maximumPoolSize，那么线程池会抛出异常RejectExecutionException。

当一个线程完成任务时，它会从队列中取下一个任务来执行。当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断。如果当前运行的线程数大于corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到corePoolSize的大小。

既然提到可以通过配置不同参数创建出不同的线程池，那么Java中默认实现好的线程池又有哪些呢？请比较它们的异同。

（1）SingleThreadExecutor线程池这个线程池只有一个核心线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- corePoolSize：1，只有一个核心线程在工作。
- maximumPoolSize：1。
- keepAliveTime：0L。
- workQueue：newLinkedBlockingQueue()，其缓冲队列是无界的。

（2）FixedThreadPool线程池

FixedThreadPool是固定大小的线程池，只有核心线程。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。FixedThreadPool多数针对一些很稳定很固定的正规并发线程，多用于服务器。

- corePoolSize：nThreads
- maximumPoolSize：nThreads
- keepAliveTime：0L
- workQueue：newLinkedBlockingQueue()，其缓冲队列是无界的。

（3）CachedThreadPool线程池CachedThreadPool是无界线程池，如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。

线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。SynchronousQueue是一个是缓冲区为1的阻塞队列。缓存型池子通常用于执行一些生存期很短的异步型任务，因此在一些面向连接的daemon型SERVER中用得不多。但对于生存期短的异步任务，它是Executor的首选。

- corePoolSize：0
- maximumPoolSize：Integer.MAX_VALUE
- keepAliveTime：60L
- workQueue：newSynchronousQueue()，一个是缓冲区为1的阻塞队列。

（4）ScheduledThreadPool线程池ScheduledThreadPool、核心线程池固定，大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。创建一个周期性执行任务的线程池。如果闲置，非核心线程池会在DEFAULT_KEEPALIVEMILLIS时间内回收。

- corePoolSize：corePoolSize
- maximumPoolSize：Integer.MAX_VALUE
- keepAliveTime：DEFAULT_KEEPALIVE_MILLIS
- workQueue：newDelayedWorkQueue()

5、如何在Java线程池中提交线程?

线程池最常用的提交任务的方法有两种：

- execute()、ExecutorService.execute方法接收一个Runable实例，它用来执行一个任务。
- submit()、ExecutorService.submit()方法返回的是Future对象。可以用isDone()来查询Future是否已经完成，当任务完成时，它具有一个结果，可以调用get()来获取结果。也可以不用isDone()进行检查就直接调用get()，在这种情况下，get()将阻塞，直至结果准备就绪。Java内存模型相关问题

什么是Java的内存模型，Java中各个线程是怎么彼此看到对方的变量的?

Java的内存模型定义了程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法参数，因为这些是线程私有的，不会被共享，所以不存在竞争问题。

Java中各个线程是怎么彼此看到对方的变量的呢?
Java中定义了主内存与工作内存的概念、所有的变量都存储在主内存，每条线程还有自己的工作内存，保存了被该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

请谈谈volatile有什么特点，为什么它能保证变量对所有线程的可见性?

关键字volatile是Java虚拟机提供的最轻量级的同步机制。当一个变量被定义成volatile之后，具备两种特性、

（1）保证此变量对所有线程的可见性。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。
（2）禁止指令重排序优化。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。Java的内存模型定义了8种内存间操作、lock和unlock把一个变量标识为一条线程独占的状态。把一个处于锁定状态的变量释放出来，释放之

后的变量才能被其他线程锁定。read和write把一个变量值从主内存传输到线程的工作内存，以便load。把store操作从工作内存得到的变量的值，放入主内存的变量中。

load和store把read操作从主内存得到的变量值放入工作内存的变量副本中。把工作内存的变量值传送到主内存，以便write。use和assgin把工作内存变量值传递给执行引擎。将执行引擎值传递给工作内存变量值。volatile的实现基于这8种内存间操作，保证了一个线程对某个volatile变量的修改，一定会被另一个线程看见，即保证了可见性。

既然volatile能够保证线程间的变量可见性，是不是就意味着基于volatile变量的运算就是并发安全的？

显然不是的。基于volatile变量的运算在并发下不一定是安全的。volatile变量在各个线程的工作内存，不存在一致性问题（各个线程的工作内存中volatile变量，每次使用前都要刷新到主内存）。但是Java里面的运算并非原子操作，导致volatile变量的运算在并发下一样是不安全的。

请对比下volatile对比Synchronized的异同。

Synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性。

ThreadLocal和Synchonized都用于解决多线程并发访问，防止任务在共享资源上产生冲突。但是ThreadLocal与Synchronized有本质的区别。

Synchronized用于实现同步机制，是利用锁的机制使变量或代码块在某一时该只能被一个线程访问，是一种"以时间换空间"的方式。而ThreadLocal为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，根除了对变量的共享，是一种"以空间换时间"的方式。

请谈谈ThreadLocal是怎么解决并发安全的？

ThreadLocal这是Java提供的一种保存线程私有信息的机制，因为其在整个线程生命周期内有效，所以可以方便地在一个线程关联的不同业务模块之间传递信息，比如事务ID、Cookie等上下文相关信息。ThreadLocal为每一个线程维护变量的副本，把共享数据的可见范围限制在同一个线程之内，其实现原理是，在ThreadLocal中有一个Map，用于存储每一个线程的变量的副本。

很多人都说要慎用ThreadLocal，谈谈你的理解，使用ThreadLocal需要注意些什么？

使用ThreadLocal要注意remove！ThreadLocal的实现是基于一个所谓的ThreadLocalMap，在ThreadLocalMap中，它的key是一个弱引用。通常弱引用都会和引用队列配合清理机制使用，但是ThreadLocal是个例外，它并没有这么做。这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应ThreadLocalMap！这就是很多OOM的来源，所以通常都会建议，应用一定要自己负责remove，并且不要和线程池配

高并发环境下构建缓存服务需要注意哪些问题？我和阿里P9聊了很久！

写在前面

> 周末，跟阿里的一个朋友（去年晋升为P9了）聊了很久，聊的内容几乎全是技术，当然了，两个技术男聊得最多的话题当然就是技术了。从基础到架构，从算法到AI，无所不谈。中间又穿插着不少天马行空的想象，虽然现在看起来不太实际，但是随着技术的进步，相信五年、十年之后都会实现的。
>
> 不知道是谁提起了在高并发环境下如何构建缓存服务，结果一路停不下来了！！

缓存特征

（1）命中率：命中数/(命中数+没有命中数)

（2）最大元素（空间）：代表缓存中可以存放的最大元素的数量，一旦缓存中元素的数量超过这个值，或者缓存数据所占的空间超过了最大支持的空间，将会触发缓存清空策略。根据不同的场景，合理设置最大元素（空间）的值，在一定程度上可以提高缓存的命中率，从而更有效的使用缓存。

（3）清空策略：FINO（先进先出）、LFU（最少使用）、LRU（最近最少使用）、过期时间、随机等。

- FINO（先进先出）：最先进入缓存的数据，在缓存空间不够或超出最大元素限制的情况下，会优先被清除掉，以腾出新的空间来接收新的数据。这种策略的算法主要是比较缓存元素的创建时间，在数据实时性较高的场景下，可以选择这种策略，优先保证最新策略可用。
- LFU（最少使用）：无论元素是否过期，根据元素的被使用次数来判断，清除使用次数最少的元素来释放空间。算法主要是比较元素的命中次数，在保证高频数据有效的场景下，可以选择这种策略。
- LRU（最近最少使用）：无论元素是否过期，根据元素最后一次被使用的时间戳，清除最远使用时间戳的元素，释放空间。算法主要是比较元素最近一次被获取的时间，在热点数据场景下，可以选择这种策略。
  过期时间：根据过期时间判断，清理过期时间最长的元素，或者清理最近要过期的元素。

缓存命中率影响因素

（1）业务场景和业务需求
缓存往往适合读多写少的场景。业务需求对实时性的要求，直接会影响到缓存的过期时间和更新策略。实时性要求越低，就越适合缓存。在相同Key和相同请求数的情况下，缓存的时间越长，命中率就会越高。

（2）缓存的设计（粒度和策略）
通常情况下，缓存的粒度越小，命中率越高。缓存的更新和命中策略也会影响缓存的命中率，当数据发生变化时，直接更新缓存的值会比移除缓存或使缓存过期的命中率更高。

（3）缓存容量和基础设施
缓存的容量有限，则容易引起缓存失效和被淘汰（目前多数的缓存框架或中间件都采用了LRU算法）。同时，缓存的技术选型也是至关重要的，比如采用应用内置的本地缓存就比较容易出现单机瓶颈，而采用分布式缓存则毕竟容易扩展。所以需要做好系统容量规划，并考虑是否可扩展。此外，不同的缓存框架或中间件，其效率和稳定性也是存在差异的。

（4）其他因素
当缓存节点发生故障时，需要避免缓存失效并最大程度降低影响，这种特殊情况也是架构师需要考虑的。业内比较典型的做法就是通过一致性Hash算法，或者通过节点冗余的方式。

有些朋友可能会有这样的理解误区：既然业务需求对数据时效性要求很高，而缓存时间又会影响到缓存命中率，那么系统就别使用缓存了。其实这忽略了一个重要因素--并发。通常来讲，在相同缓存时间和key的情况下，并发越高，缓存的收益会越高，即便缓存时间很短。

提高缓存命中率的方法

从架构师的角度，需要应用尽可能的通过缓存直接获取数据，并避免缓存失效。这也是比较考验架构师能力的，需要在业务需求，缓存粒度，缓存策略，技术选型等各个方面去通盘考虑并做权衡。尽可能的聚焦在高频访问且时效性要求不高的热点业务上，通过缓存预加载（预热）、增加存储容量、调整缓存粒度、更新缓存等手段来提高命中率。

对于时效性很高（或缓存空间有限），内容跨度很大（或访问很随机），并且访问量不高的应用来说缓存命中率可能长期很低，可能预热后的缓存还没来得被访问就已经过期了。

缓存的分类和应用场景

（1）本地缓存：编程实现（成员变量、局部变量、静态变量）、Guava Cache
（2）分布式缓存：Memcached、Redis

高并发场景下缓存常见问题

（1）缓存的一致性
更新数据库成功——更新缓存失败
更新缓存成功——更新数据库失败
更新数据库成功——淘汰缓存失败
淘汰缓存成功——更新数据库失败

（2）缓存并发
并发时请求缓存时已过期或者没有命中或者更新的情况下有大量的请求访问数据库。

解决办法：在缓存更新或者过期的情况下,先尝试获取到lock,当更新完成后,尝试释放锁,其他的请求只需要牺牲一定的等待时间

（3）缓存穿透
在高并发的场景下,如果某一个key被高并发的访问没有被命中,出于对容错性的考虑会尝试从后端的数据库获取，从而导致大量的请求访问了数据库,主要是当key对应的数据为空或者为null的情况下，这就导致数据库中并发的执行了很多不必要的查询操作。从而导致了巨大的冲击和压力。

解决方法:
缓存空对象：对查询结果为空的对象也进行缓存，如果是集合可以缓存一个空的集合，而不是null,如果是单个对象可以通过字段标识来区分,需要保证缓存数据的时效性(实现相对简单)，适合命中不高但可能会频繁更新的数据。
单独过滤处理：对所有可能对应数据为空的key进行统一的存放,并在请求前做拦截(实现相对复杂)，适合命中不高更新不频繁的数据

（4）缓存颠簸问题
缓存的颠簸问题，有些地方可能被称为“缓存抖动”，可以看作是一种比“雪崩”更轻微的故障，但是也会在一段时间内对系统造成冲击和性能影响。一般是由于缓存节点故障导致。业内推荐的做法是通过一致性Hash算法来解决。

（5）缓存雪崩现象
缓存雪崩就是指由于缓存的原因，导致大量请求到达后端数据库，从而导致数据库崩溃，整个系统崩溃，发生灾难。导致这种现象的原因有很多种，上面提到的“缓存并发”，“缓存穿透”，“缓存颠簸”等问题，其实都可能会导致缓存雪崩现象发生。这些问题也可能会被恶意攻击者所利用。还有一种情况，例如某个时间点内，系统预加载的缓存周期性集中失效了，也可能会导致雪崩。为了避免这种周期性失效，可以通过设置不同的过期时间，来错开缓存过期，从而避免缓存集中失效。

从应用架构角度，我们可以通过限流、降级、熔断等手段来降低影响，也可以通过多级缓存来避免这种灾难。

此外，从整个研发体系流程的角度，应该加强压力测试，尽量模拟真实场景，尽早的暴露问题从而防范。

（6）缓存无底洞现象

该问题由 facebook 的工作人员提出的， facebook 在 2010 年左右，memcached 节点就已经达3000 个，缓存数千 G 内容。他们发现了一个问题---memcached 连接频率，效率下降了，于是加 memcached 节点，添加了后，发现因为连接频率导致的问题，仍然存在，并没有好转，称之为"无底洞现象"

系统架构篇

高并发秒杀系统架构解密，不是所有的秒杀都是秒杀！

前言

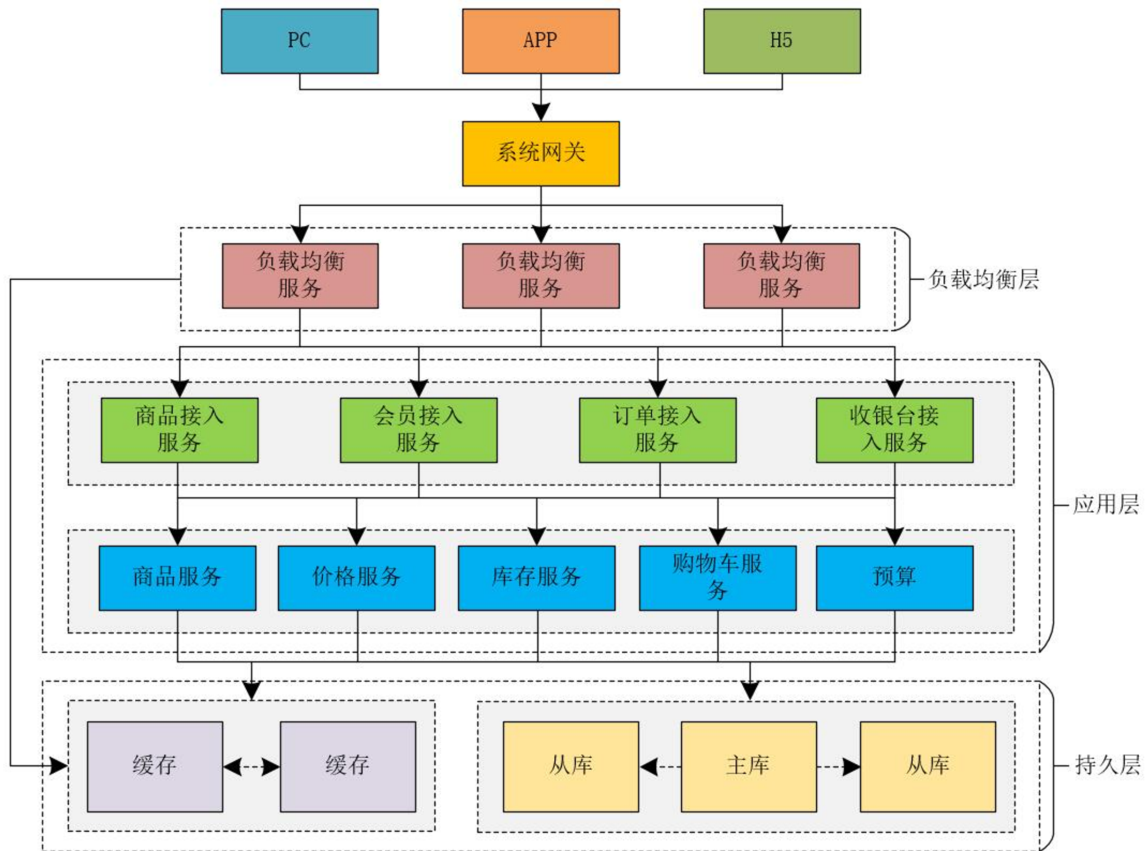
> 很多小伙伴反馈说，高并发专题学了那么久，但是，在真正做项目时，仍然不知道如何下手处理高并发业务场景！甚至很多小伙伴仍然停留在只是简单的提供接口（CRUD）阶段，不知道学习的并发知识如何运用到实际项目中，就更别提如何构建高并发系统了！

究竟什么样的系统算是高并发系统？今天，我们就一起解密高并发业务场景下典型的秒杀系统的架构，结合高并发专题下的其他文章，学以致用。

电商系统架构

在电商领域，存在着典型的秒杀业务场景，那何谓秒杀场景呢。简单的来说就是一件商品的购买人数远远大于这件商品的库存，而且这件商品在很短的时间内就会被抢购一空。 比如每年的618、双11大促，小米新品促销等业务场景，就是典型的秒杀业务场景。

我们可以将电商系统的架构简化成下图所示。



由图所示，我们可以简单的将电商系统的核心层分为：负载均衡层、应用层和持久层。接下来，我们就预估下每一层的并发量。

- 假如负载均衡层使用的是高性能的Nginx，则我们可以预估Nginx最大的并发度为：10W+，这里是以万为单位。
- 假设应用层我们使用的是Tomcat，而Tomcat的最大并发度可以预估为800左右，这里是以百为单位。
- 假设持久层的缓存使用的是Redis，数据库使用的是MySQL，MySQL的最大并发度可以预估为1000左右，以千为单位。Redis的最大并发度可以预估为5W左右，以万为单位。

所以，负载均衡层、应用层和持久层各自的并发度是不同的，那么，为了提升系统的总体并发度和缓存，我们通常可以采取哪些方案呢？

（1）系统扩容

系统扩容包括垂直扩容和水平扩容，增加设备和机器配置，绝大多数的场景有效。

（2）缓存

本地缓存或者集中式缓存，减少网络IO，基于内存读取数据。大部分场景有效。

（3）读写分离

采用读写分离，分而治之，增加机器的并行处理能力。

秒杀系统的特点

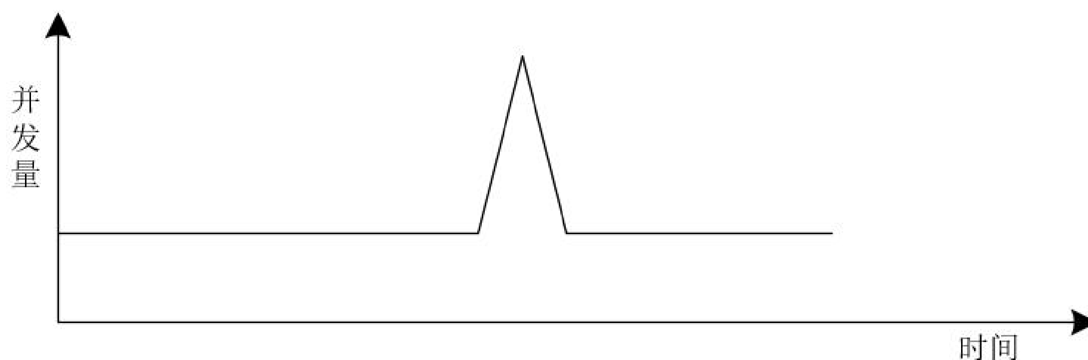
对于秒杀系统来说，我们可以从**业务和技术**两个角度来阐述其自身存在的一些特点。

秒杀系统的业务特点

这里，我们可以使用12306网站来举例，每年春运时，12306网站的访问量是非常大的，但是网站平时的访问量却是比较平缓的，也就是说，每年春运时节，12306网站的访问量会出现瞬时突增的现象。

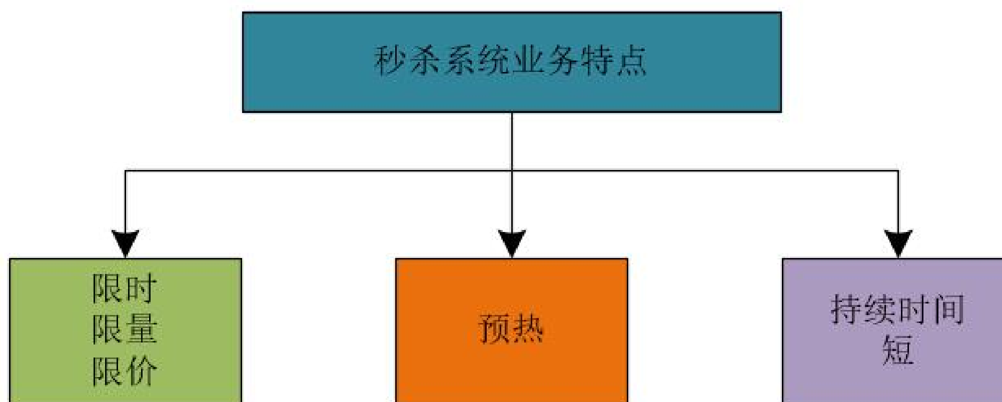
再比如，小米秒杀系统，在上午10点开售商品，10点前的访问量比较平缓，10点时同样会出现并发量瞬时突增的现象。

所以，秒杀系统的流量和并发量我们可以使用下图来表示。



由图可以看出，秒杀系统的并发量存在瞬时凸峰的特点，也叫做流量突刺现象。

我们可以将秒杀系统的特点总结如下。



（1）限时、限量、限价

在规定的时间内进行；秒杀活动中商品的数量有限；商品的价格会远远低于原来的价格，也就是说，在秒杀活动中，商品会以远远低于原来的价格出售。

例如，秒杀活动的时间仅限于某天上午10点到10点半，商品数量只有10万件，售完为止，而且商品的价格非常低，例如：1元购等业务场景。

限时、限量和限价可以单独存在，也可以组合存在。

（2）活动预热

需要提前配置活动；活动还未开始时，用户可以查看活动的相关信息；秒杀活动开始前，对活动进行大力宣传。

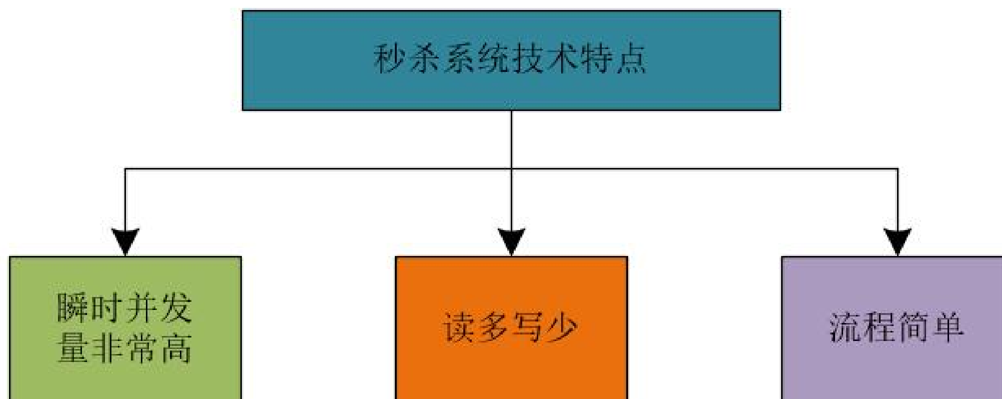
（3）持续时间短

购买的人数数量庞大；商品会迅速售完。

在系统流量呈现上，就会出现一个突刺现象，此时的并发访问量是非常高的，大部分秒杀场景下，商品会在极短的时间内售完。

秒杀系统的技术特点

我们可以将秒杀系统的技术特点总结如下。



（1）瞬时并发量非常高

大量用户会在同一时间抢购商品；瞬间并发峰值非常高。

（2）读多写少

系统中商品页的访问量巨大；商品的可购买数量非常少；库存的查询访问数量远远大于商品的购买数量。

在商品页中往往会加入一些限流措施，例如早期的秒杀系统商品页会加入验证码来平滑前端对系统的访问流量，近期的秒杀系统商品详情页会在用户打开页面时，提示用户登录系统。这都是对系统的访问进行限流的一些措施。

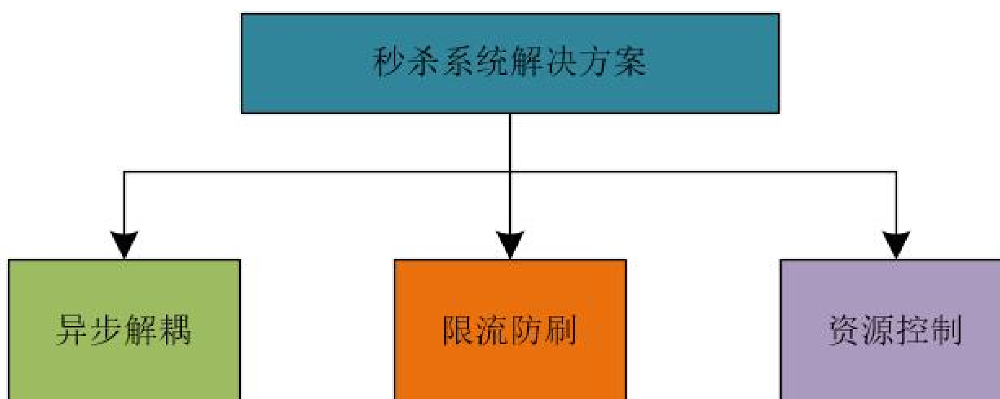
（3）流程简单

秒杀系统的业务流程一般比较简单；总体上来说，秒杀系统的业务流程可以概括为：下单减库存。

针对这种短时间内大流量的系统来说，就不太适合使用系统扩容了，因为即使系统扩容了，也就是在很短的时间内会使用到扩容后的系统，大部分时间内，系统无需扩容即可正常访问。 那么，我们可以采取哪些方案来提升系统的秒杀性能呢?

秒杀系统方案

针对秒杀系统的特点，我们可以采取如下的措施来提升系统的性能。



（1）异步解耦

将整体流程进行拆解，核心流程通过队列方式进行控制。

（2）限流防刷

控制网站整体流量，提高请求的门槛，避免系统资源耗尽。

（3）资源控制

将整体流程中的资源调度进行控制，扬长避短。

由于应用层能够承载的并发量比缓存的并发量少很多。所以，在高并发系统中，我们可以直接使用OpenResty由负载均衡层访问缓存，避免了调用应用层的性能损耗。大家可以到https://openresty.org/cn/来了解有关OpenResty更多的知识。同时，由于秒杀系统中，商品数量比较少，我们也可以使用动态渲染技术，CDN技术来加速网站的访问性能。

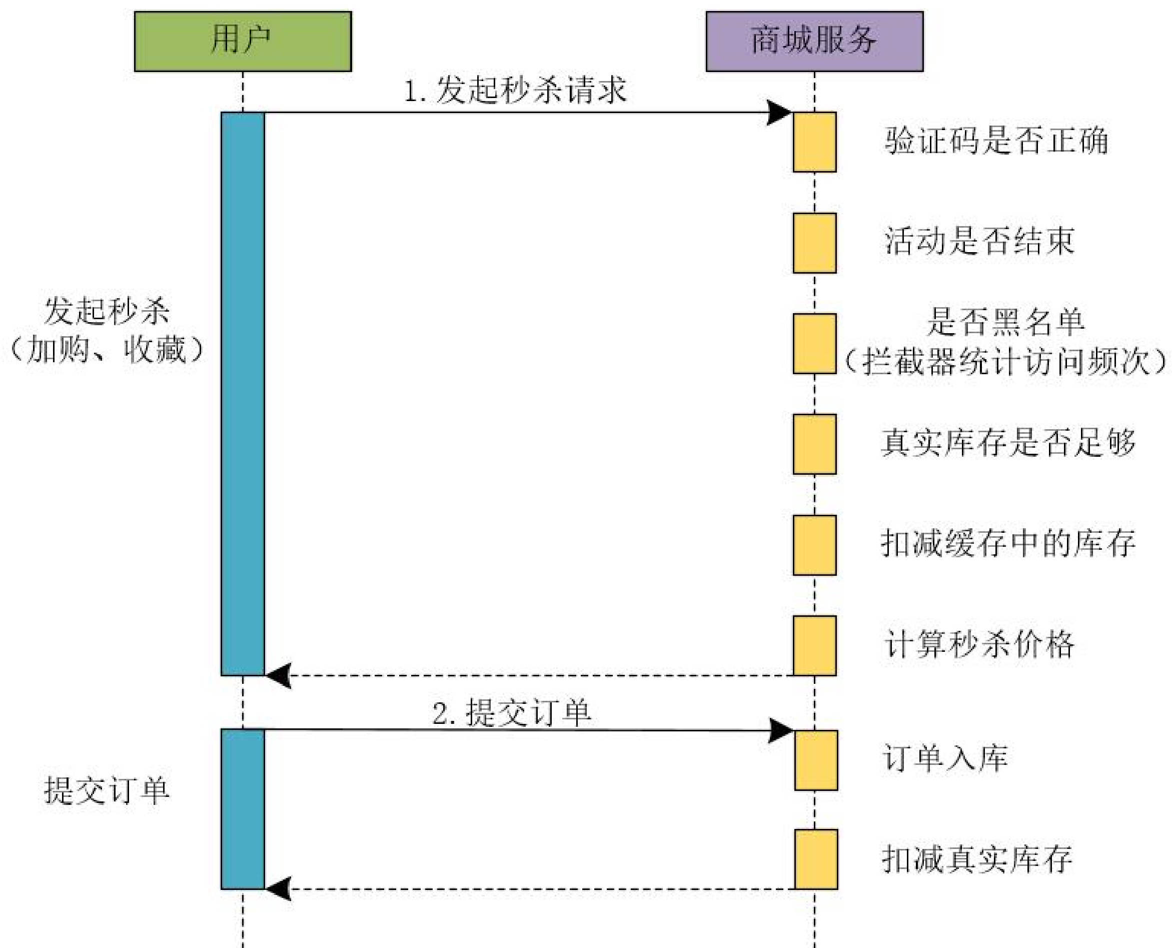
如果在秒杀活动开始时，并发量太高时，我们可以将用户的请求放入队列中进行处理，并为用户弹出排队页面。

注：图片来自魅族

秒杀系统时序图

网上很多的秒杀系统和对秒杀系统的解决方案，并不是真正的秒杀系统，他们采用的只是同步处理请求的方案，一旦并发量真的上来了，他们所谓的秒杀系统的性能会急剧下降。我们先来看一下秒杀系统在同步下单时的时序图。

同步下单流程



1.用户发起秒杀请求

在同步下单流程中，首先，用户发起秒杀请求。商城服务需要依次执行如下流程来处理秒杀请求的业务。

（1）识别验证码是否正确

商城服务判断用户发起秒杀请求时提交的验证码是否正确。

（2）判断活动是否已经结束

验证当前秒杀活动是否已经结束。

（3）验证访问请求是否处于黑名单

在电商领域中，存在着很多的恶意竞争，也就是说，其他商家可能会通过不正当手段来恶意请求秒杀系统，占用系统大量的带宽和其他系统资源。此时，就需要使用风控系统等实现黑名单机制。为了简单，也可以使用拦截器统计访问频次实现黑名单机制。

（4）验证真实库存是否足够

系统需要验证商品的真实库存是否足够，是否能够支持本次秒杀活动的商品库存量。

（5）扣减缓存中的库存

在秒杀业务中，往往会将商品库存等信息存放在缓存中，此时，还需要验证秒杀活动使用的商品库存是否足够，并且需要扣减秒杀活动的商品库存数量。

（6）计算秒杀的价格

由于在秒杀活动中，商品的秒杀价格和商品的真实价格存在差异，所以，需要计算商品的秒杀价格。

注意：如果在秒杀场景中，系统涉及的业务更加复杂的话，会涉及更多的业务操作，这里，我只是列举出一些常见的业务操作。

2.提交订单

（1）订单入口

将用户提交的订单信息保存到数据库中。

（2）扣减真实库存

订单入库后，需要在商品的真实库存中将本次成功下单的商品数量扣除。

如果我们使用上述流程开发了一个秒杀系统，当用户发起秒杀请求时，由于系统每个业务流程都是串行执行的，整体上系统的性能不会太高，当并发量太高时，我们会为用户弹出下面的排队页面，来提示用户进行等待。

注：图片来自魅族

此时的排队时间可能是15秒，也可能是30秒，甚至是更长时间。这就存在一个问题：在用户发起秒杀请求到服务器返回结果的这段时间内，客户端和服务器之间的连接不会被释放，这就会占大量占用服务器的资源。

网上很多介绍如何实现秒杀系统的文章都是采用的这种方式，那么，这种方式能做秒杀系统吗？答案是可以做，但是这种方式支撑的并发量并不是太高。此时，有些网友可能会问：我们公司就是这样做的秒杀系统啊！上线后一直在用，没啥问题啊！我想说的是：使用同步下单方式确实可以做秒杀系统，但是同步下单的性能不会太高。之所以你们公司采用同步下单的方式做秒杀系统没出现大的问题，那是因为你们的秒杀系统的并发量没达到一定的量级，也就是说，你们的秒杀系统的并发量其实并不高。

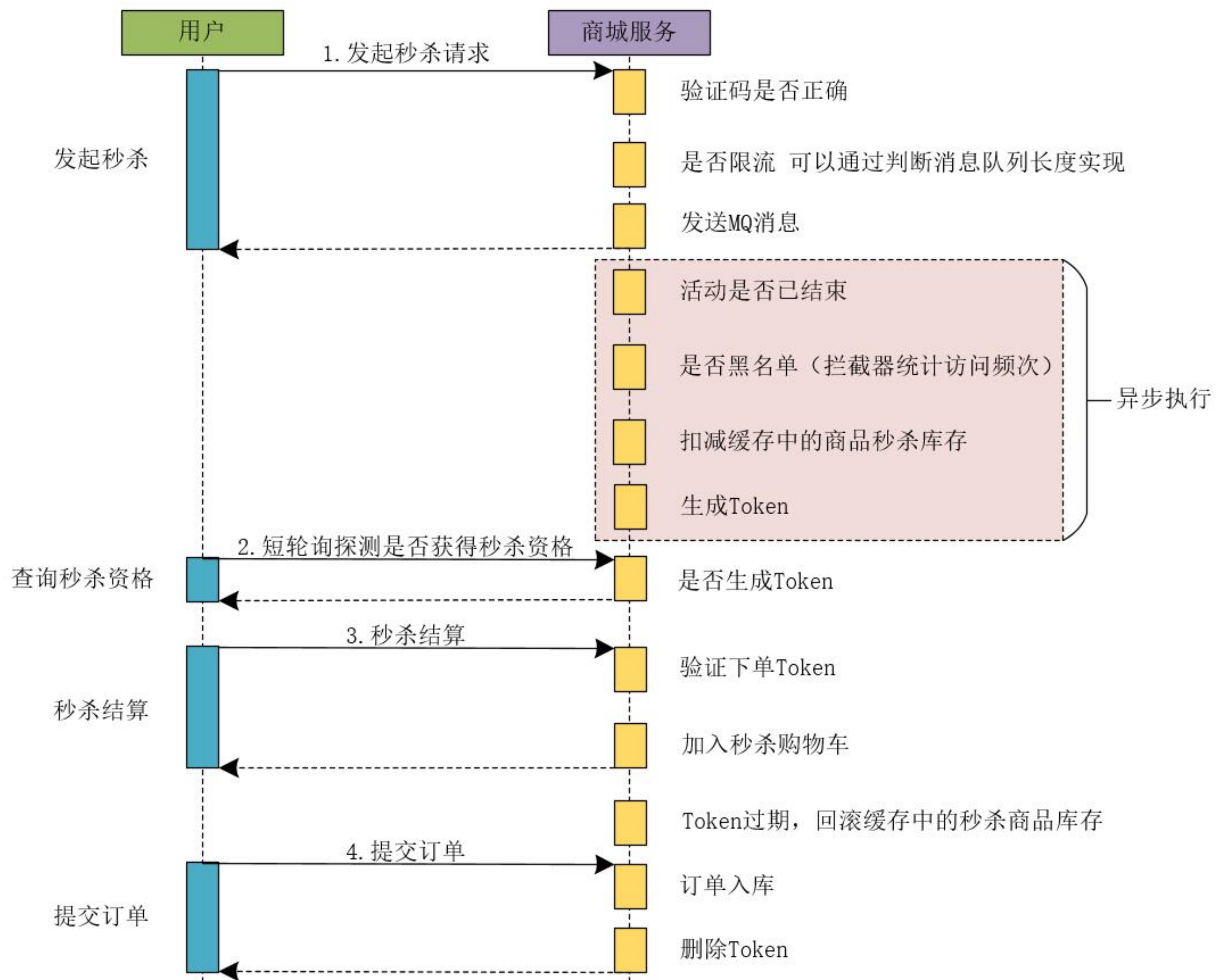
所以，很多所谓的秒杀系统，存在着秒杀的业务，但是称不上真正的秒杀系统，原因就在于他们使用的是同步的下单流程，限制了系统的并发流量。之所以上线后没出现太大的问题，是因为系统的并发量不高，不足以压死整个系统。

如果12306、淘宝、天猫、京东、小米等大型商城的秒杀系统是这么玩的话，那么，他们的系统迟早会被玩死，他们的系统工程师不被开除才怪！所以，在秒杀系统中，这种同步处理下单的业务流程的方案是不可取的。

以上就是同步下单的整个流程操作，如果下单流程更加复杂的话，就会涉及到更多的业务操作。

异步下单流程

既然同步下单流程的秒杀系统称不上真正的秒杀系统，那我们就需要采用异步的下单流程了。异步的下单流程不会限制系统的高并发流量。



1.用户发起秒杀请求

用户发起秒杀请求后，商城服务会经过如下业务流程。

（1）检测验证码是否正确

用户发起秒杀请求时，会将验证码一同发送过来，系统会检验验证码是否有效，并且是否正确。

（2）是否限流

系统会对用户的请求进行是否限流的判断，这里，我们可以通过判断消息队列的长度来进行判断。因为我们将用户的请求放在了消息队列中，消息队列中堆积的是用户的请求，我们可以根据当前消息队列中存在的待处理的请求数量来判断是否需要对用户的请求进行限流处理。

例如，在秒杀活动中，我们出售1000件商品，此时在消息队列中存在1000个请求，如果后续仍然有用户发起秒杀请求，则后续的请求我们可以不再处理，直接向用户返回商品已售完的提示。

所以，使用限流后，我们可以更快的处理用户的请求和释放连接的资源。

（3）发送MQ

用户的秒杀请求通过前面的验证后，我们就可以将用户的请求参数等信息发送到MQ中进行异步处理，同时，向用户响应结果信息。在商城服务中，会有专门的异步任务处理模块来消费消息队列中的请求，并处理后续的异步流程。

在用户发起秒杀请求时，异步下单流程比同步下单流程处理的业务操作更少，它将后续的操作通过MQ发送给异步处理模块进行处理，并迅速向用户返回响应结果，释放请求连接。

2.异步处理

我们可以将下单流程的如下操作进行异步处理。

（1）判断活动是否已经结束

（2）判断本次请求是否处于系统黑名单，为了防止电商领域同行的恶意竞争可以为系统增加黑名单机制，将恶意的请求放入系统的黑名单中。可以使用拦截器统计访问频次来实现。

（3）扣减缓存中的秒杀商品的库存数量。

（4）生成秒杀Token，这个Token是绑定当前用户和当前秒杀活动的，只有生成了秒杀Token的请求才有资格进行秒杀活动。

这里我们引入了异步处理机制，**在异步处理中，系统使用多少资源，分配多少线程来处理相应的任务，是可以进行控制的。**

3.短轮询查询秒杀结果

这里，可以采取客户端短轮询查询是否获得秒杀资格的方案。例如，客户端可以每隔3秒钟轮询请求服务器，查询是否获得秒杀资格，这里，我们在服务器的处理就是判断当前用户是否存在秒杀Token，如果服务器为当前用户生成了秒杀Token，则当前用户存在秒杀资格。否则继续轮询查询，直到超时或者服务器返回商品已售完或者无秒杀资格等信息为止。

采用短轮询查询秒杀结果时，在页面上我们同样可以提示用户排队处理中，但是此时客户端会每隔几秒轮询服务器查询秒杀资格的状态，相比于同步下单流程来说，无需长时间占用请求连接。

此时，**可能会有网友会问：采用短轮询查询的方式，会不会存在直到超时也查询不到是否具有秒杀资格的状态呢？答案是：有可能！** 这里我们试想一下秒杀的真实场景，商家参加秒杀活动本质上不是为了赚钱，而是提升商品的销量和商家的知名度，吸引更多的用户来买自己的商品。所以，我们不必保证用户能够100%的查询到是否具有秒杀资格的状态。

4.秒杀结算

（1）验证下单Token

客户端提交秒杀结算时，会将秒杀Token一同提交到服务器，商城服务会验证当前的秒杀Token是否有效。

（2）加入秒杀购物车

商城服务在验证秒杀Token合法并有效后，会将用户秒杀的商品添加到秒杀购物车。

5.提交订单

（1）订单入库

将用户提交的订单信息保存到数据库中。

（2）删除Token

秒杀商品订单入库成功后，删除秒杀Token。

这里大家可以思考一个问题：我们为什么只在异步下单流程的粉色部分采用异步处理，而没有在其他部分采取异步削峰和填谷的措施呢？

这是因为在异步下单流程的设计中，无论是在产品设计上还是在接口设计上，我们在用户发起秒杀请求阶段对用户的请求进行了限流操作，可以说，系统的限流操作是非常前置的。在用户发起秒杀请求时进行了限流，系统的高峰流量已经被平滑解决了，再往后走，其实系统的并发量和系统流量并不是非常高了。

所以，网上很多的文章和帖子中在介绍秒杀系统时，说是在下单时使用异步削峰来进行一些限流操作，那都是在扯淡！ 因为下单操作在整个秒杀系统的流程中属于比较靠后的操作了，限流操作一定要前置处理，在秒杀业务后面的流程中做限流操作是没啥卵用的。

高并发“黑科技”与致胜奇招

假设，在秒杀系统中我们使用Redis实现缓存，假设Redis的读写并发量在5万左右。我们的商城秒杀业务需要支持的并发量在100万左右。如果这100万的并发全部打入Redis中，Redis很可能就会挂掉，那么，我们如何解决这个问题呢？接下来，我们就一起来探讨这个问题。

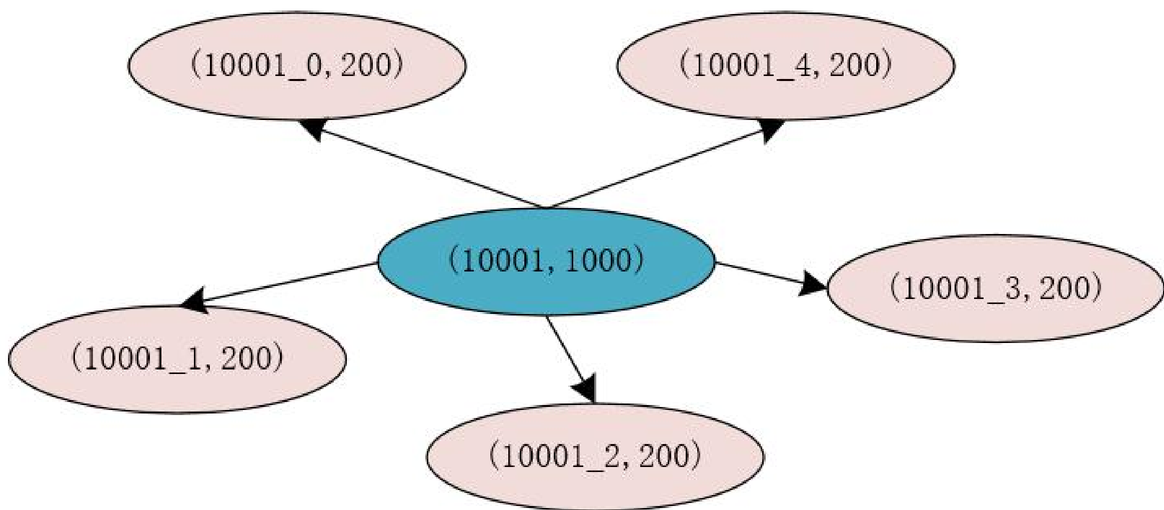
> 在高并发的秒杀系统中，如果采用Redis缓存数据，则Redis缓存的并发处理能力是关键，因为很多的前缀操作都需要访问Redis。而异步削峰只是基本的操作，关键还是要保证Redis的并发处理能力。

解决这个问题的关键思想就是：分而治之，将商品库存分开放。

暗度陈仓

我们在Redis中存储秒杀商品的库存数量时，可以将秒杀商品的库存进行“分割”存储来提升Redis的读写并发量。

例如，原来的秒杀商品的id为10001，库存为1000件，在Redis中的存储为(10001, 1000)，我们将原有的库存分割为5份，则每份的库存为200件，此时，我们在Redia中存储的信息为(10001_0, 200)，(10001_1, 200)，(10001_2, 200)，(10001_3, 200)，(10001_4, 200)。



此时，我们将库存进行分割后，每个分割后的库存使用商品id加上一个数字标识来存储，这样，在对存储商品库存的每个Key进行Hash运算时，得出的Hash结果是不同的，这就说明，存储商品库存的Key有很大概率不在Redis的同一个槽位中，这就能够提升Redis处理请求的性能和并发量。

分割库存后，我们还需要在Redis中存储一份商品id和分割库存后的Key的映射关系，此时映射关系的Key为商品的id，也就是10001，Value为分割库存后存储库存信息的Key，也就是10001_0，10001_1，10001_2，10001_3，10001_4。在Redis中我们可以使用List来存储这些值。

在真正处理库存信息时，我们可以先从Redis中查询出秒杀商品对应的分割库存后的所有Key，同时使用AtomicLong来记录当前的请求数量，使用请求数量对从Redia中查询出的秒杀商品对应的分割库存后的所有Key的长度进行求模运算，得出的结果为0，1，2，3，4。再在前面拼接上商品id就可以得出真正的库存缓存的Key。此时，就可以根据这个Key直接到Redis中获取相应的库存信息。

移花接木

在高并发业务场景中，我们可以直接使用Lua脚本库（OpenResty）从负载均衡层直接访问缓存。

这里，我们思考一个场景：如果在秒杀业务场景中，秒杀的商品被瞬间抢购一空。此时，用户再发起秒杀请求时，如果系统由负载均衡层请求应用层的各个服务，再由应用层的各个服务访问缓存和数据库，其实，本质上已经没有任何意义了，因为商品已经卖完了，再通过系统的应用层进行层层校验已经没有太多意义了！！ 而应用层的并发访问量是以百为单位的，这又在一定程度上会降低系统的并发度。

为了解决这个问题，此时，**我们可以在系统的负载均衡层取出用户发送请求时携带的用户id，商品id和秒杀活动id等信息，直接通过Lua脚本等技术来访问缓存中的库存信息。如果秒杀商品的库存小于或者等于0，则直接返回用户商品已售完的提示信息，而不用再经过应用层的层层校验了。** 针对这个架构，我们可以参见本文中的电商系统的架构图（正文开始的第一张图）。

写在最后

> 如果觉得文章对你有点帮助，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发编程技术。

高并发分布式锁架构解密，不是所有的锁都是分布式锁！！

写在前面

> 最近，很多小伙伴留言说，在学习高并发编程时，不太明白分布式锁是用来解决什么问题的，还有不少小伙伴甚至连分布式锁是什么都不太明白。明明在生产环境上使用了自己开发的分布式锁，为什么还会出现问题呢？同样的程序，加上分布式锁后，性能差了几个数量级！这又是为什么呢？今天，我们就来说说如何在高并发环境下实现分布式锁，不是所有的锁都是高并发的。
>
> 万字长文，带你深入解密高并发环境下的分布式锁架构，不是所有的锁都是分布式锁！！！

究竟什么样的锁才能更好的支持高并发场景呢？今天，我们就一起解密高并发环境下典型的分布式锁架构，结合【高并发】专题下的其他文章，学以致用。

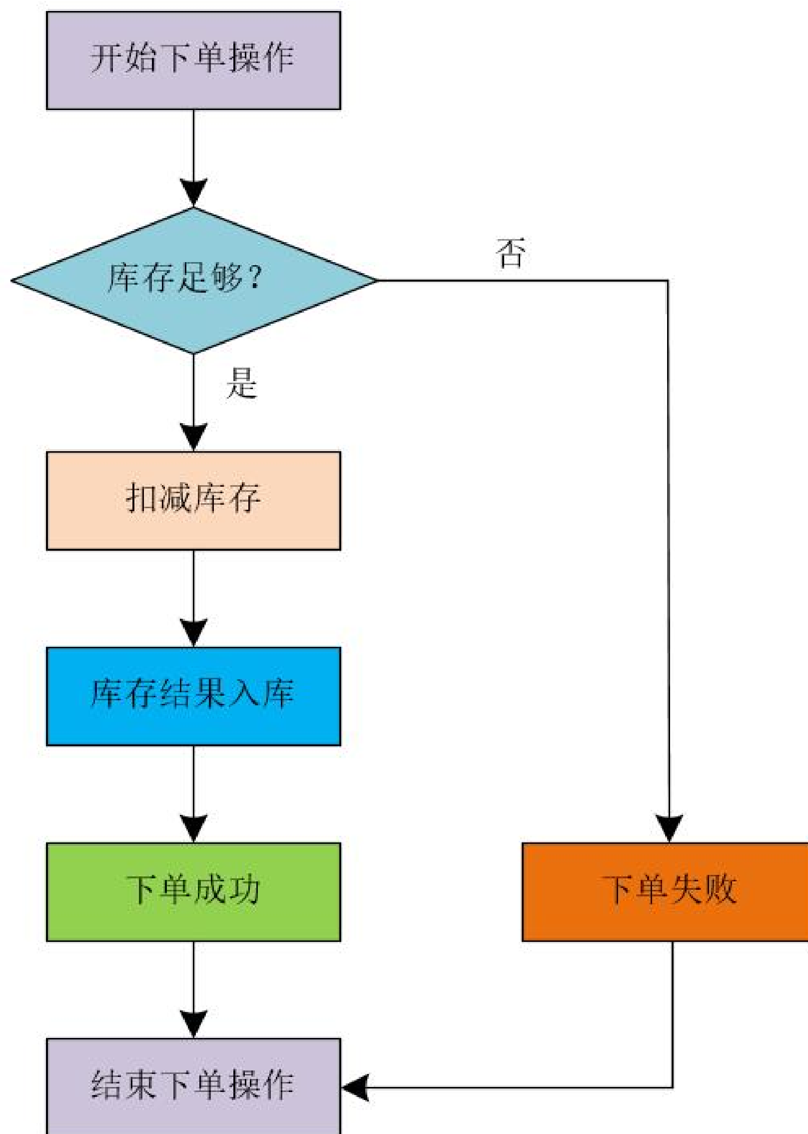
锁用来解决什么问题呢?

在我们编写的应用程序或者高并发程序中，不知道大家有没有想过一个问题，就是我们为什么需要引入锁？锁为我们解决了什么问题呢？

在很多业务场景下，我们编写的应用程序中会存在很多的 **资源竞争** 的问题。而我们在高并发程序中，引入锁，就是为了解决这些资源竞争的问题。

电商超卖问题

这里，我们可以列举一个简单的业务场景。比如，在电子商务（商城）的业务场景中，提交订单购买商品时，首先需要查询相应商品的库存是否足够，只有在商品库存数量足够的前提下，才能让用户成功的下单。下单时，我们需要在库存数量中减去用户下单的商品数量，并将库存操作的结果数据更新到数据库中。整个流程我们可以简化成下图所示。



很多小伙伴也留言说，让我给出代码，这样能够更好的学习和掌握相关的知识。好吧，这里，我也给出相应的代码片段吧。我们可以使用下面的代码片段来表示用户的下单操作，我这里将商品的库存信息保存在了Redis中。

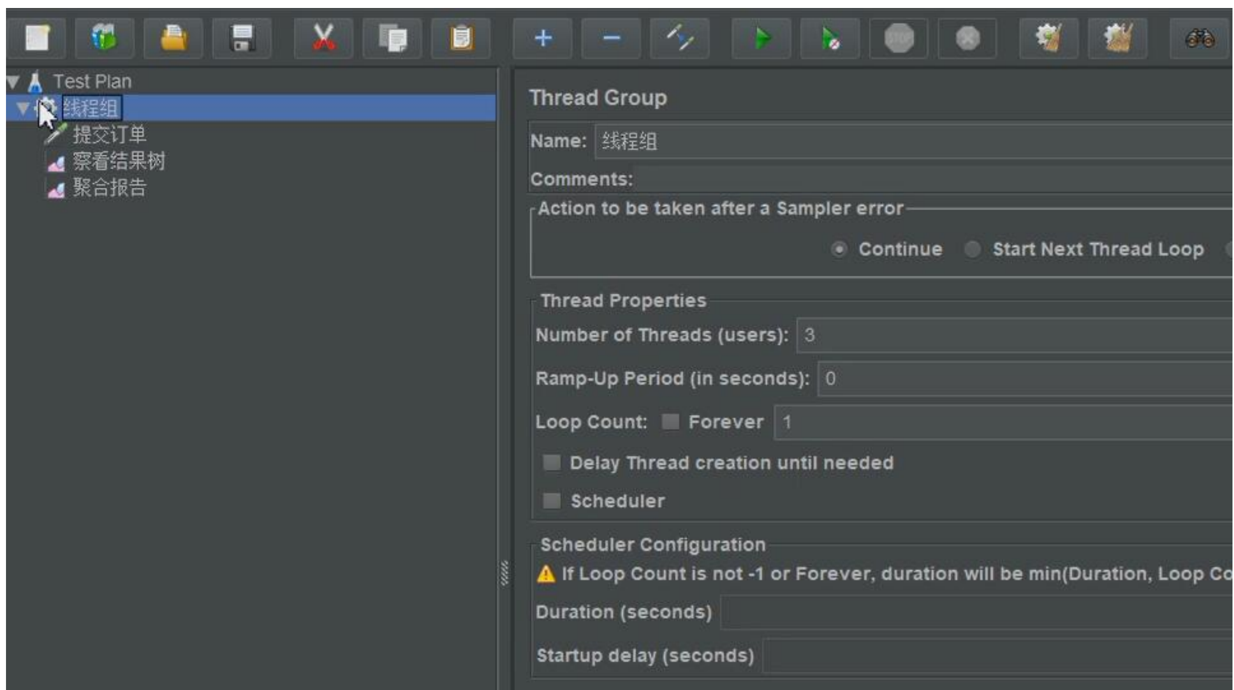
```
@RequestMapping("/submitOrder")
public String submitOrder(){
    int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock"));
    if(stock > 0){
        stock -= 1;
        stringRedisTemplate.opsForValue().set("stock", String.valueOf(stock));
        logger.debug("库存扣减成功，当前库存为：{}", stock);
    }else{
        logger.debug("库存不足，扣减库存失败");
        throw new OrderException("库存不足，扣减库存失败");
    }
    return "success";
}
```

注意：上述代码片段比较简单，只是为了方便大家理解，真正项目中的代码就不能这么写了。

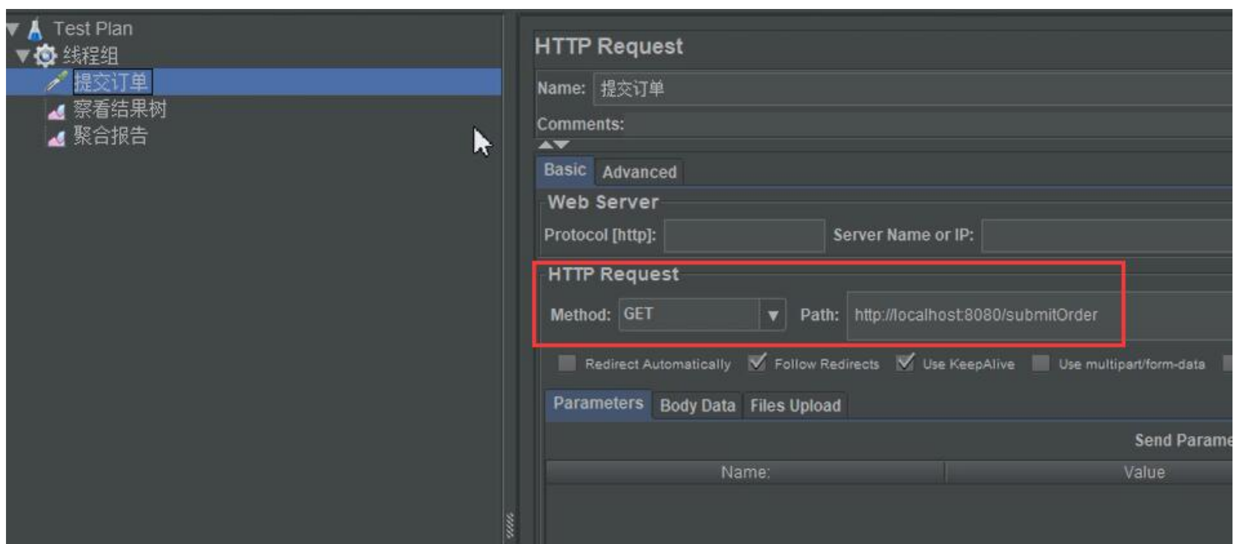
上述的代码看似是没啥问题的，但是我们不能只从代码表面上来观察代码的执行顺序。这是因为在JVM中代码的执行顺序未必是按照我们书写代码的顺序执行的。即使在JVM中代码是按照我们书写的顺序执行，那我们对外提供的接口一旦暴露出去，就会有成千上万的客户端来访问我们的接口。所以说，我们暴露出去的接口是会被并发访问的。

试问，上面的代码在高并发环境下是线程安全的吗？答案肯定不是线程安全的，因为上述扣减库存的操作会出现并行执行的情况。

我们可以使用Apache JMeter来对上述接口进行测试，这里，我使用Apache JMeter对上述接口进行测试。



在Jmeter中，我将线程的并发度设置为3，接下来的配置如下所示。



以HTTP GET请求的方式来并发访问提交订单的接口。此时，运行JMeter来访问接口，命令行会打印出下面的日志信息。

```
库存扣减成功，当前库存为：49
库存扣减成功，当前库存为：49
库存扣减成功，当前库存为：49
```

这里，我们明明请求了3次，也就是说，提交了3笔订单，为什么扣减后的库存都是一样的呢？这种现象在电商领域有一个专业的名词叫做 **"超卖"**。

如果一个大型的高并发电商系统，比如淘宝、天猫、京东等，出现了超卖现象，那损失就无法估量了！架构设计和开发电商系统的人员估计就要通通下岗了。所以，**作为技术人员，我们一定要严谨的对待技术，严格做好系统的每一个技术环节。**

JVM中提供的锁

JVM中提供的synchronized和Lock锁，相信大家并不陌生了，很多小伙伴都会使用这些锁，也能使用这些锁来实现一些简单的线程互斥功能。那么，作为立志要成为架构师的你，是否了解过JVM锁的底层原理呢？

JVM锁原理

说到JVM锁的原理，我们就不得不限说说Java中的对象头了。

Java中的对象头

> 每个Java对象都有对象头。如果是非数组类型，则用2个字宽来存储对象头，如果是数组，则会用3个字宽来存储对象头。在32位处理器中，一个字宽是32位；在64位虚拟机中，一个字宽是64位。

对象头的内容如下表 。

长度	内容	说明
32/64bit	Mark Word	存储对象的hashCode或锁信息等
32/64bit	Class Metadata Access	存储到对象类型数据的指针
32/64bit	Array length	数组的长度（如果是数组）

Mark Work的格式如下所示。

锁状态	29bit或61bit	1bit是否是偏向锁?	2bit锁标志位
无锁		0	01
偏向锁	线程ID	1	01
轻量级锁	指向栈中锁记录的指针	此时这一位不用于标识偏向锁	00
重量级锁	指向互斥量（重量级锁）的指针	此时这一位不用于标识偏向锁	10
GC标记		此时这一位不用于标识偏向锁	11

可以看到，当对象状态为偏向锁时， Mark Word 存储的是偏向的线程ID；当状态为轻量级锁时， Mark Word 存储的是指向线程栈中 Lock Record 的指针；当状态为重量级锁时， Mark Word 为指向堆中的monitor对象的指针 。

> 有关Java对象头的知识，参考《深入浅出Java多线程》。

JVM锁原理

简单点来说，JVM中锁的原理如下。

在Java对象的对象头上，有一个锁的标记，比如，第一个线程执行程序时，检查Java对象头中的锁标记，发现Java对象头中的锁标记为未加锁状态，于是为Java对象进行了加锁操作，将对象头中的锁标记设置为锁定状态。第二个线程执行同样的程序时，也会检查Java对象头中的锁标记，此时会发现Java对象头中的锁标记的状态为锁定状态。于是，第二个线程会进入相应的阻塞队列中进行等待。

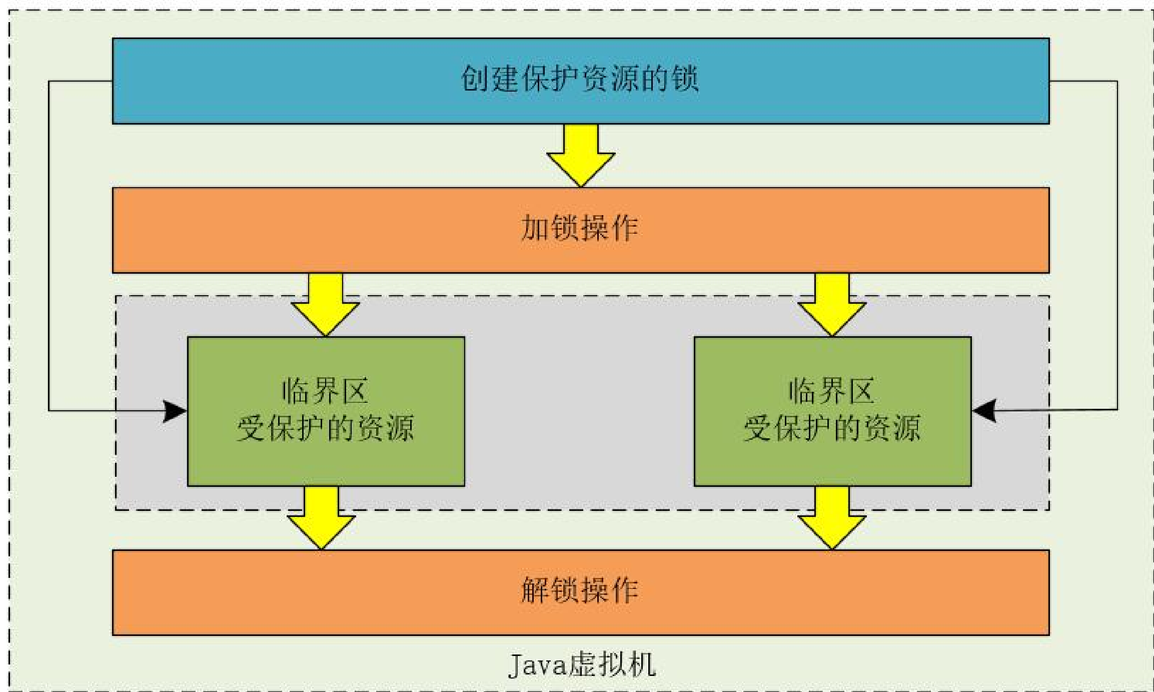
这里有一个关键点就是Java对象头中的锁标记如何实现。

JVM锁的短板

JVM中提供的synchronized和Lock锁都是JVM级别的，大家都知道，当运行一个Java程序时，会启动一个JVM进程来运行我们的应用程序。synchronized和Lock在JVM级别有效，也就是说，synchronized和Lock在同一Java进程内有效。如果我们开发的应用程序是分布式的，那么只是使用synchronized和Lock来解决分布式场景下的高并发问题，就会显得有点力不从心了。

synchronized和Lock支持JVM同一进程内部的线程互斥

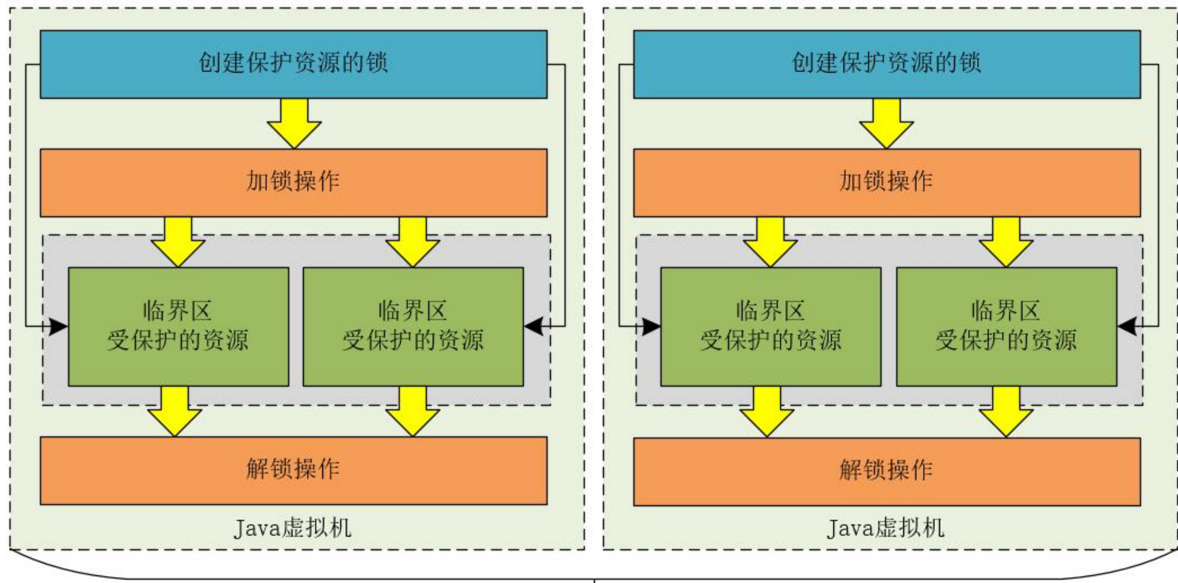
synchronized和Lock在JVM级别能够保证高并发程序的互斥，我们可以使用下图来表示。



但是，当我们将应用程序部署成分布式架构，或者将应用程序在不同的JVM进程中运行时，synchronized和Lock就不能保证分布式架构和多JVM进程下应用程序的互斥性了。

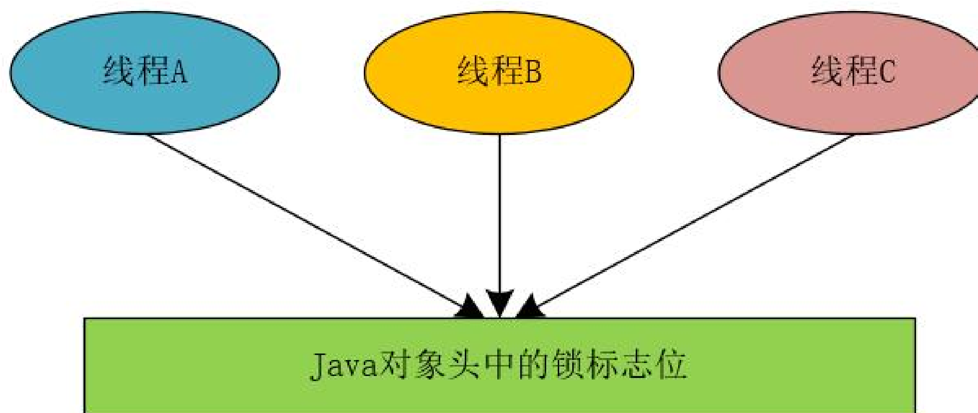
synchronized和Lock不能实现多JVM进程之间的线程互斥

分布式架构和多JVM进程的本质都是将应用程序部署在不同的JVM实例中，也就是说，其本质还是多JVM进程。

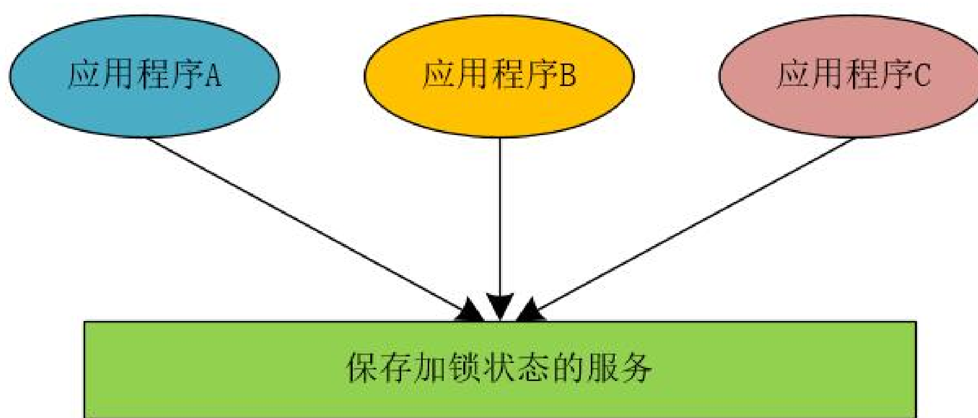


分布式锁

我们在实现分布式锁时，可以参照JVM锁实现的思想，JVM锁在为对象加锁时，通过改变Java对象的对象头中的锁的标志位来实现，也就是说，所有的线程都会访问这个Java对象的对象头中的锁标志位。



我们同样以这种思想来实现分布式锁，当我们将应用程序进行拆分并部署成分布式架构时，所有应用程序中的线程访问共享变量时，都到同一个地方去检查当前程序的临界区是否进行了加锁操作，而是否进行了加锁操作，我们在统一的地方使用相应的状态来进行标记。



可以看到，在分布式锁的实现思想上，与JVM锁相差不大。而在实现分布式锁中，**保存加锁状态的服务可以使用MySQL、Redis和Zookeeper实现。**

但是，在互联网高并发环境中， **使用Redis实现分布式锁的方案是使用的最多的。** 接下来，我们就使用Redis来深入解密分布式锁的架构设计。

Redis如何实现分布式锁

Redis命令

在Redis中，有一个不常使用的命令如下所示。

```
SETNX key value
```

这条命令的含义就是“SET if Not Exists”，即不存在的时候才会设置值。

只有在key不存在的情况下，将键key的值设置为value。如果key已经存在，则SETNX命令不做任何操作。

这个命令的返回值如下。

- 命令在设置成功时返回1。
- 命令在设置失败时返回0。

所以，我们在分布式高并发环境下，可以使用Redis的SETNX命令来实现分布式锁。假设此时有线程A和线程B同时访问临界区代码，假设线程A首先执行了SETNX命令，并返回结果1，继续向下执行。而此时线程B再次执行SETNX命令时，返回的结果为0，则线程B不能继续向下执行。只有当线程A执行DELETE命令将设置的锁状态删除时，线程B才会成功执行SETNX命令设置加锁状态后继续向下执行。

引入分布式锁

了解了如何使用Redis中的命令实现分布式锁后，我们就可以对下单接口进行改造了，加入分布式锁，如下所示。

```
/**
```

```
* 为了演示方便，我这里就简单定义了一个常量作为商品的id
* 实际工作中，这个商品id是前端进行下单操作传递过来的参数
*/
public static final String PRODUCT_ID = "100001";

@RequestMapping("/submitOrder")
public String submitOrder(){
    //通过stringRedisTemplate来调用Redis的SETNX命令，key为商品的id，value为字符串“binghe”
    //实际上，value可以为任意的字符换
    Boolean isLocked = stringRedisTemplate.opsForValue().setIfAbsent(PRODUCT_ID, "binghe");
   //没有拿到锁，返回下单失败
    if(!isLock){
        return "failure";
    }
    int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock"));
    if(stock > 0){
        stock -= 1;
        stringRedisTemplate.opsForValue().set("stock", String.valueOf(stock));
        logger.debug("库存扣减成功，当前库存为：{}", stock);
    }else{
        logger.debug("库存不足，扣减库存失败");
        throw new OrderException("库存不足，扣减库存失败");
    }
    //业务执行完成，删除PRODUCT_ID key
    stringRedisTemplate.delete(PRODUCT_ID);
    return "success";
}
```

那么，在上述代码中，我们加入了分布式锁的操作，那上述代码是否能够在高并发场景下保证业务的原子性呢？答案是可以保证业务的原子性。但是，**在实际场景中，上面实现分布式锁的代码是不可用的！！**

假设当线程A首先执行stringRedisTemplate.opsForValue()的setIfAbsent()方法返回true，继续向下执行，正在执行业务代码时，抛出了异常，线程A直接退出了JVM。此时，stringRedisTemplate.delete(PRODUCT_ID);代码还没来得及执行，之后所有的线程进入提交订单的方法时，调用stringRedisTemplate.opsForValue()的setIfAbsent()方法都会返回false。导致后续的所有下单操作都会失败。**这就是分布式场景下的死锁问题。**

所以，上述代码中实现分布式锁的方式在实际场景下是不可取的！！

引入try-finally代码块

说到这，相信小伙伴们都能够想到，使用try-finall代码块啊，接下来，我们为下单接口的方法加上try-finally代码块。

```
/**
* 为了演示方便，我这里就简单定义了一个常量作为商品的id
* 实际工作中，这个商品id是前端进行下单操作传递过来的参数
*/
public static final String PRODUCT_ID = "100001";

@RequestMapping("/submitOrder")
public String submitOrder(){
    //通过stringRedisTemplate来调用Redis的SETNX命令，key为商品的id，value为字符串“binghe”
    //实际上，value可以为任意的字符换
    Boolean isLocked = stringRedisTemplate.opsForValue().setIfAbsent(PRODUCT_ID, "binghe");
   //没有拿到锁，返回下单失败
    if(!isLock){
        return "failure";
    }
    try{
        int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock"));
        if(stock > 0){
            stock -= 1;
            stringRedisTemplate.opsForValue().set("stock", String.valueOf(stock));
            logger.debug("库存扣减成功，当前库存为：{}", stock);
        }else{
            logger.debug("库存不足，扣减库存失败");
            throw new OrderException("库存不足，扣减库存失败");
        }
    }finally{
         //业务执行完成，删除PRODUCT_ID key
        stringRedisTemplate.delete(PRODUCT_ID);
    }
```

```
    return "success";
}
```

那么，上述代码是否真正解决了死锁的问题呢？我们在写代码时，不能只盯着代码本身，觉得上述代码没啥问题了。实际上，生产环境是非常复杂的。如果线程在成功加锁之后，执行业务代码时，还没来得及执行删除锁标志的代码，此时，服务器宕机了，程序并没有优雅的退出JVM。也会使得后续的线程进入提交订单的方法时，因无法成功的设置锁标志位而下单失败。所以说，上述的代码仍然存在问题。

引入Redis超时机制

在Redis中可以设置缓存的自动过期时间，我们可以将其引入到分布式锁的实现中，如下代码所示。

```
/**
* 为了演示方便，我这里就简单定义了一个常量作为商品的id
* 实际工作中，这个商品id是前端进行下单操作传递过来的参数
*/
public static final String PRODUCT_ID = "100001";

@RequestMapping("/submitOrder")
public String submitOrder(){
    //通过stringRedisTemplate来调用Redis的SETNX命令，key为商品的id，value为字符串“binghe”
    //实际上，value可以为任意的字符换
    Boolean isLocked = stringRedisTemplate.opsForValue().setIfAbsent(PRODUCT_ID, "binghe");
   //没有拿到锁，返回下单失败
    if(!isLock){
        return "failure";
    }
    try{
        stringRedisTemplate.expire(PRODUCT_ID, 30, TimeUnit.SECONDS);
        int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock"));
        if(stock > 0){
            stock -= 1;
            stringRedisTemplate.opsForValue().set("stock", String.valueOf(stock));
            logger.debug("库存扣减成功，当前库存为：{}", stock);
        }else{
            logger.debug("库存不足，扣减库存失败");
            throw new OrderException("库存不足，扣减库存失败");
        }
    }finally{
         //业务执行完成，删除PRODUCT_ID key
        stringRedisTemplate.delete(PRODUCT_ID);
    }
    return "success";
}
```

在上述代码中，我们加入了如下一行代码来为Redis中的锁标志设置过期时间。

```
stringRedisTemplate.expire(PRODUCT_ID, 30, TimeUnit.SECONDS);
```

此时，我们设置的过期时间为30秒。

那么问题来了，这样是否就真正的解决了问题呢？上述程序就真的没有坑了吗？**答案是还是有坑的！！**

“坑位”分析

我们在下单操作的方法中为分布式锁引入了超时机制，此时的代码还是无法真正避免死锁的问题，那“坑位”到底在哪里呢？试想，当程序执行完stringRedisTemplate.opsForValue().setIfAbsent()方法后，正要执行stringRedisTemplate.expire(PRODUCT_ID, 30, TimeUnit.SECONDS)代码时，服务器宕机了，你还别说，生产坏境的情况非常复杂，就是这么巧，服务器就宕机了。此时，后续请求进入提交订单的方法时，都会因为无法成功设置锁标志而导致后续下单流程无法正常执行。

既然我们找到了上述代码的“坑位”，那我们如何将这个”坑“填上？如何解决这个问题呢？别急，Redis已经提供了这样的功能。我们可以在向Redis中保存数据的时候，可以同时指定数据的超时时间。所以，我们可以将代码改造成如下所示。

```
/**
* 为了演示方便，我这里就简单定义了一个常量作为商品的id
* 实际工作中，这个商品id是前端进行下单操作传递过来的参数
*/
public static final String PRODUCT_ID = "100001";
```

```
@RequestMapping("/submitOrder")
public String submitOrder(){
    //通过stringRedisTemplate来调用Redis的SETNX命令，key为商品的id，value为字符串“binghe”
    //实际上，value可以为任意的字符换
    Boolean isLocked = stringRedisTemplate.opsForValue().setIfAbsent(PRODUCT_ID, "binghe", 30,
TimeUnit.SECONDS);
   //没有拿到锁，返回下单失败
    if(!isLock){
        return "failure";
    }
    try{
        int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock"));
        if(stock > 0){
            stock -= 1;
            stringRedisTemplate.opsForValue().set("stock", String.valueOf(stock));
            logger.debug("库存扣减成功，当前库存为：{}", stock);
        }else{
            logger.debug("库存不足，扣减库存失败");
            throw new OrderException("库存不足，扣减库存失败");
        }
    }finally{
         //业务执行完成，删除PRODUCT_ID key
        stringRedisTemplate.delete(PRODUCT_ID);
    }
    return "success";
}
```

在上述代码中，我们在向Redis中设置锁标志位的时候就设置了超时时间。此时，只要向Redis中成功设置了数据，则即使我们的业务系统宕机，Redis中的数据过期后，也会自动删除。后续的线程进入提交订单的方法后，就会成功的设置锁标志位，并向下执行正常的下单流程。

到此，上述的代码基本上在功能角度解决了程序的死锁问题，那么，上述程序真的就完美了吗？哈哈，很多小伙伴肯定会说不完美！确实，上面的代码还不是完美的，那大家知道哪里不完美吗？接下来，我们继续分析。

在开发集成角度分析代码

在我们开发公共的系统组件时，比如我们这里说的分布式锁，我们肯定会抽取一些公共的类来完成相应的功能来供系统使用。

这里，假设我们定义了一个RedisLock接口，如下所示。

```
public interface RedisLock{
    //加锁操作
    boolean tryLock(String key, long timeout, TimeUnit unit);
    //解锁操作
    void releaseLock(String key);
}
```

接下来，使用RedisLockImpl类实现RedisLock接口，提供具体的加锁和解锁实现，如下所示。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        return stringRedisTemplate.opsForValue().setIfAbsent(key, "binghe", timeout, unit);
    }
    @Override
    public void releaseLock(String key){
        stringRedisTemplate.delete(key);
    }
}
```

在开发集成的角度来说，当一个线程从上到下执行时，首先对程序进行加锁操作，然后执行业务代码，执行完成后，再进行释放锁的操作。理论上，加锁和释放锁时，操作的Redis Key都是一样的。但是，如果其他开发人员在编写代码时，并没有调用tryLock()方法，而是直接调用了releaseLock()方法，并且他调用releaseLock()方法传递的key与你调用tryLock()方法传递的key是一样的。那此时就会出现问题了，他在编写代码时，硬生生的将你加的锁释放了！！！

所以，上述代码是不安全的，别人能够随随便便的将你加的锁删除，这就是锁的误删操作，这是非常危险的，所以，上述的程序存在很严重的问题！！

那如何实现只有加锁的线程才能进行相应的解锁操作呢？ 继续向下看。

如何实现加锁和解锁的归一化？

什么是加锁和解锁的归一化呢？简单点来说，就是一个线程执行了加锁操作后，后续必须由这个线程执行解锁操作，加锁和解锁操作由同一个线程来完成。

为了解决只有加锁的线程才能进行相应的解锁操作的问题，那么，我们就需要将加锁和解锁操作绑定到同一个线程中，那么，如何将加锁操作和解锁操作绑定到同一个线程呢？其实很简单，相信很多小伙伴都想到了—— **使用ThreadLocal实现** 。没错，使用ThreadLocal类确实能够解决这个问题。

此时，我们将RedisLockImpl类的代码修改成如下所示。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        String uuid = UUID.randomUUID().toString();
        threadLocal.set(uuid);
        return stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的key与Redis中的Key相同时，在执行删除锁的操作
        if(threadLocal.get().equals(stringRedisTemplate.opsForValue().get(key))){
            stringRedisTemplate.delete(key);
        }
    }
}
```

上述代码的主要逻辑为：在对程序执行尝试加锁操作时，首先生成一个uuid，将生成的uuid绑定到当前线程，并将传递的key参数操作Redis中的key，生成的uuid作为Redis中的Value，保存到Redis中，同时设置超时时间。当执行解锁操作时，首先，判断当前线程中绑定的uuid是否和Redis中存储的uuid相等，只有二者相等时，才会执行删除锁标志位的操作。这就避免了一个线程对程序进行了加锁操作后，其他线程对这个锁进行了解锁操作的问题。

继续分析

我们将加锁和解锁的方法改成如下所示。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;
    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();
    private String lockUUID;
    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        String uuid = UUID.randomUUID().toString();
        threadLocal.set(uuid);
        lockUUID = uuid;
        return stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的key与Redis中的Key相同时，在执行删除锁的操作
        if(lockUUID.equals(stringRedisTemplate.opsForValue().get(key))){
            stringRedisTemplate.delete(key);
        }
    }
}
```

相信很多小伙伴都会看出上述代码存在什么问题了！！ 没错，那就是 **线程安全的问题。**

所以，这里，我们需要使用ThreadLocal来解决线程安全问题。

可重入性分析

在上面的代码中，当一个线程成功设置了锁标志位后，其他的线程再设置锁标志位时，就会返回失败。还有一种场景就是在提交订单的接口方法中，调用了服务A，服务A调用了服务B，而服务B的方法中存在对同一个商品的加锁和解锁操作。

所以，服务B成功设置锁标志位后，提交订单的接口方法继续执行时，也不能成功设置锁标志位了。也就是说，目前实现的分布式锁没有可重入性。

这里，就存在可重入性的问题了。我们希望设计的分布式锁 **具有可重入性** ，那什么是可重入性呢？简单点来说，就是同一个线程，能够多次获取同一把锁，并且能够按照顺序进行解决操作。

其实，在JDK 1.5之后提供的锁很多都支持可重入性，比如synchronized和Lock。

如何实现可重入性呢?

映射到我们加锁和解锁方法时，我们如何支持同一个线程能够多次获取到锁（设置锁标志位）呢？可以这样简单的设计：如果当前线程没有绑定uuid，则生成uuid绑定到当前线程，并且在Redis中设置锁标志位。如果当前线程已经绑定了uuid，则直接返回true，证明当前线程之前已经设置了锁标志位，也就是说已经获取到了锁，直接返回true。

结合以上分析，我们将提交订单的接口方法代码改造成如下所示。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        Boolean isLocked = false;
        if(threadLocal.get() == null){
            String uuid = UUID.randomUUID().toString();
            threadLocal.set(uuid);
            isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
        }else{
            isLocked = true;
        }
        return isLocked;
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的key与Redis中的Key相同时，在执行删除锁的操作
        if(threadLocal.get().equals(stringRedisTemplate.opsForValue().get(key))){
            stringRedisTemplate.delete(key);
        }
    }
}
```

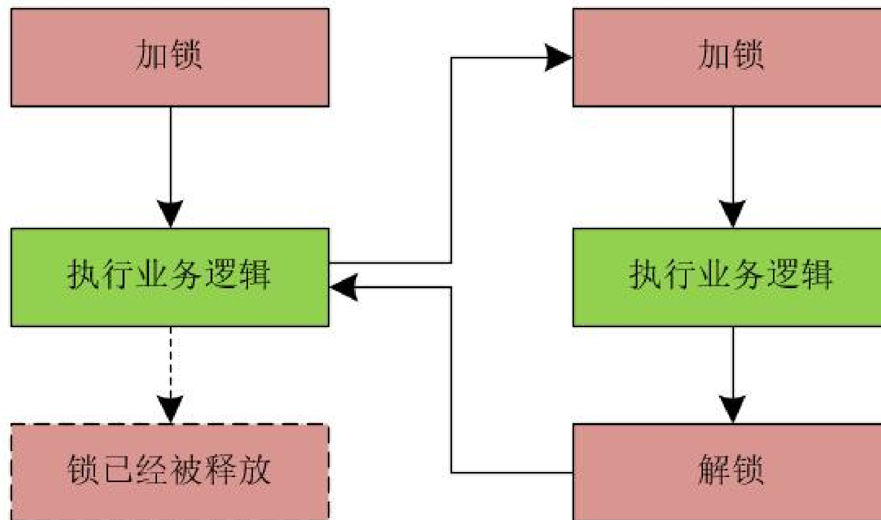
这样写看似没有啥问题，但是大家细想一下，这样写就真的OK了吗？

可重入性的问题分析

既然上面分布式锁的可重入性是存在问题的，那我们就来分析下问题的根源在哪里！

假设我们提交订单的方法中，首先使用RedisLock接口对代码块添加了分布式锁，在加锁后的代码中调用了服务A，而服务A中也存在调用RedisLock接口的加锁和解锁操作。而多次调用RedisLock接口的加锁操作时，只要之前的锁没有失效，则会直接返回true，表示成功获取锁。也就是说，无论调用加锁操作多少次，最终只会成功加锁一次。而执行完服务A中的逻辑后，在服务A中调用RedisLock接口的解锁方法，此时，会将当前线程所有的加锁操作获得的锁全部释放掉。

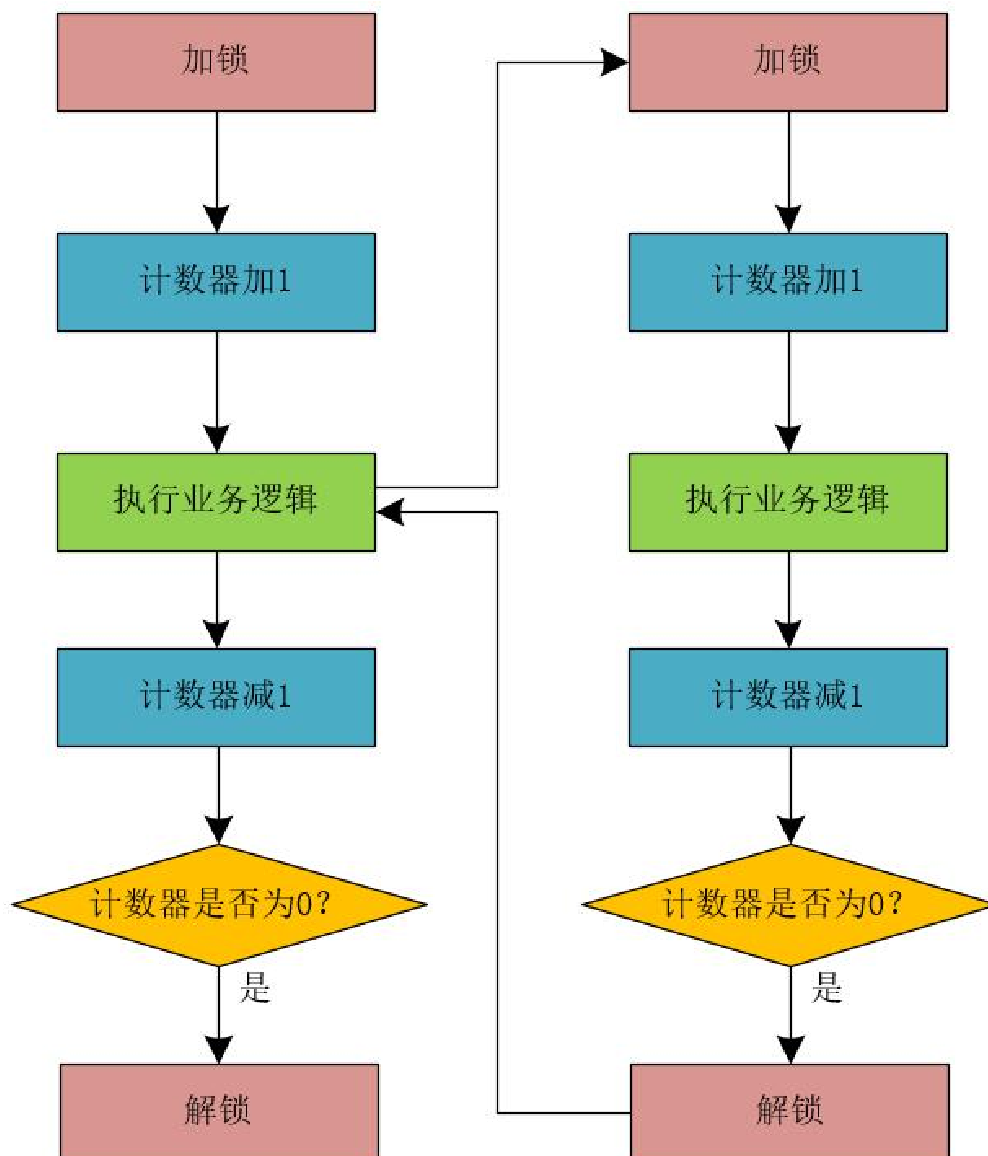
我们可以使用下图来简单的表示这个过程。



那么问题来了，如何解决可重入性的问题呢?

解决可重入性问题

相信很多小伙伴都能够想出使用计数器的方式来解决上面可重入性的问题，没错，**就是使用计数器来解决。** 整体流程如下所示。



那么，体现在程序代码上是什么样子呢？我们来修改RedisLockImpl类的代码，如下所示。

```java
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    private ThreadLocal<Integer> threadLocalInteger = new ThreadLocal<Integer>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        Boolean isLocked = false;
        if(threadLocal.get() == null){
            String uuid = UUID.randomUUID().toString();
            threadLocal.set(uuid);
            isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
        }else{
            isLocked = true;
        }
        //加锁成功后将计数器加1
        if(isLocked){
            Integer count = threadLocalInteger.get() == null ? 0 : threadLocalInteger.get();
            threadLocalInteger.set(count++);
        }
        return isLocked;
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的key与Redis中的Key相同时，再执行删除锁的操作
        if(threadLocal.get().equals(stringRedisTemplate.opsForValue().get(key))){
            Integer count = threadLocalInteger.get();
            //计数器减为0时释放锁
            if(count == null || --count <= 0){
                stringRedisTemplate.delete(key);
            }
        }
    }
}
```

至此，我们基本上解决了分布式锁的可重入性问题。

说到这里，我还要问大家一句，**上面的解决问题的方案真的没问题了吗？**

阻塞与非阻塞锁

在提交订单的方法中，当获取Redis分布式锁失败时，我们直接返回了failure来表示当前请求下单的操作失败了。试想，在高并发环境下，一旦某个请求获得了分布式锁，那么，在这个请求释放锁之前，其他的请求调用下单方法时，都会返回下单失败的信息。在真实场景中，这是非常不友好的。我们可以将后续的请求进行阻塞，直到当前请求释放锁后，再唤醒阻塞的请求获得分布式锁来执行方法。

所以，我们设计的分布式锁需要支持 **阻塞和非阻塞** 的特性。

那么，如何实现阻塞呢？我们可以使用自旋来实现，继续修改RedisLockImpl的代码如下所示。

```java
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    private ThreadLocal<Integer> threadLocalInteger = new ThreadLocal<Integer>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        Boolean isLocked = false;
        if(threadLocal.get() == null){
            String uuid = UUID.randomUUID().toString();
            threadLocal.set(uuid);
            isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
            //如果获取锁失败，则自旋获取锁，直到成功
            if(!isLocked){
                for(;;){
```

```
                    isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
                    if(isLocked){
                        break;
                    }
                }
            }
        }else{
            isLocked = true;
        }
        //加锁成功后将计数器加1
        if(isLocked){
            Integer count = threadLocalInteger.get() == null ? 0 : threadLocalInteger.get();
            threadLocalInteger.set(count++);
        }
        return isLocked;
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的key与Redis中的Key相同时，再执行删除锁的操作
        if(threadLocal.get().equals(stringRedisTemplate.opsForValue().get(key))){
            Integer count = threadLocalInteger.get();
            //计数器减为0时释放锁
            if(count == null || --count <= 0){
                stringRedisTemplate.delete(key);
            }
        }
    }
}
```

在分布式锁的设计中，**阻塞锁和非阻塞锁** 是非常重要的概念，大家一定要记住这个知识点。

锁失效问题

尽管我们实现了分布式锁的阻塞特性，但是还有一个问题是我们不得不考虑的。那就是 **锁失效** 的问题。

当程序执行业务的时间超过了锁的过期时间会发生什么呢？ 想必很多小伙伴都能够想到，那就是前面的请求没执行完，锁过期失效了，后面的请求获取到分布式锁，继续向下执行了，程序无法做到真正的互斥，无法保证业务的原子性了。

那如何解决这个问题呢？**答案就是：我们必须保证在业务代码执行完毕后，才能释放分布式锁。** 方案是有了，那如何实现呢？

说白了，我们需要在业务代码中，时不时的执行下面的代码来保证在业务代码没执行完时，分布式锁不会因超时而被释放。

```
springRedisTemplate.expire(PRODUCT_ID, 30, TimeUnit.SECONDS);
```

这里，我们需要定义一个定时策略来执行上面的代码，需要注意的是：我们不能等到30秒后再执行上述代码，因为30秒时，锁已经失效了。例如，我们可以每10秒执行一次上面的代码。

有些小伙伴说，直接在RedisLockImpl类中添加一个while(true)循环来解决这个问题，那我们就这样修改下RedisLockImpl类的代码，看看有没有啥问题。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    private ThreadLocal<Integer> threadLocalInteger = new ThreadLocal<Integer>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        Boolean isLocked = false;
        if(threadLocal.get() == null){
            String uuid = UUID.randomUUID().toString();
            threadLocal.set(uuid);
            isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
            //如果获取锁失败，则自旋获取锁，直到成功
            if(!isLocked){
                for(;;){
                    isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
                    if(isLocked){
                        break;
```

```
                    }
                }
            }
            //定义更新锁的过期时间
            while(true){
                Integer count = threadLocalInteger.get();
                //当前锁已经被释放，则退出循环
                if(count == 0 || count <= 0){
                    break;
                }
                springRedisTemplate.expire(key, 30, TimeUnit.SECONDS);
                try{
                    //每隔10秒执行一次
                    Thread.sleep(10000);
                }catch (InterruptedException e){
                    e.printStackTrace();
                }
            }
        }else{
            isLocked = true;
        }
        //加锁成功后将计数器加1
        if(isLocked){
            Integer count = threadLocalInteger.get() == null ? 0 : threadLocalInteger.get();
            threadLocalInteger.set(count++);
        }
        return isLocked;
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的uuid与Redis中的uuid相同时，再执行删除锁的操作
        if(threadLocal.get().equals(stringRedisTemplate.opsForValue().get(key))){
            Integer count = threadLocalInteger.get();
            //计数器减为0时释放锁
            if(count == null || --count <= 0){
                stringRedisTemplate.delete(key);
            }
        }
    }
}
```

相信小伙伴们看了代码就会发现哪里有问题了：更新锁过期时间的代码肯定不能这么去写。因为这么写会 **导致当前线程在更新锁超时时间的while(true)循环中一直阻塞而无法返回结果。** 所以，我们不能将当前线程阻塞，需要异步执行定时任务来更新锁的过期时间。

此时，我们继续修改RedisLockImpl类的代码，将定时更新锁超时的代码放到一个单独的线程中执行，如下所示。

```
public class RedisLockImpl implements RedisLock{
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    private ThreadLocal<Integer> threadLocalInteger = new ThreadLocal<Integer>();

    @Override
    public boolean tryLock(String key, long timeout, TimeUnit unit){
        Boolean isLocked = false;
        if(threadLocal.get() == null){
            String uuid = UUID.randomUUID().toString();
            threadLocal.set(uuid);
            isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
            //如果获取锁失败，则自旋获取锁，直到成功
            if(!isLocked){
                for(;;){
                    isLocked = stringRedisTemplate.opsForValue().setIfAbsent(key, uuid, timeout, unit);
                    if(isLocked){
                        break;
                    }
                }
```

```
            }
            //启动新线程来执行定时任务，更新锁过期时间
           new Thread(new UpdateLockTimeoutTask(uuid, stringRedisTemplate, key)).start();
        }else{
            isLocked = true;
        }
        //加锁成功后将计数器加1
        if(isLocked){
            Integer count = threadLocalInteger.get() == null ? 0 : threadLocalInteger.get();
            threadLocalInteger.set(count++);
        }
        return isLocked;
    }
    @Override
    public void releaseLock(String key){
        //当前线程中绑定的uuid与Redis中的uuid相同时，再执行删除锁的操作
        String uuid = stringRedisTemplate.opsForValue().get(key);
        if(threadLocal.get().equals(uuid)){
            Integer count = threadLocalInteger.get();
            //计数器减为0时释放锁
            if(count == null || --count <= 0){
                stringRedisTemplate.delete(key);
                //获取更新锁超时时间的线程并中断
                long threadId = stringRedisTemplate.opsForValue().get(uuid);
                Thread updateLockTimeoutThread = ThreadUtils.getThreadByThreadId(threadId);
                if(updateLockTimeoutThread != null){
                     //中断更新锁超时时间的线程
                    updateLockTimeoutThread.interrupt();
                    stringRedisTemplate.delete(uuid);
                }
            }
        }
    }
}
```

创建UpdateLockTimeoutTask类来执行更新锁超时的时间。

```
public class UpdateLockTimeoutTask implements Runnable{
    //uuid
    private long uuid;
    private StringRedisTemplate stringRedisTemplate;
    private String key;
    public UpdateLockTimeoutTask(long uuid, StringRedisTemplate stringRedisTemplate, String key){
        this.uuid = uuid;
        this.stringRedisTemplate = stringRedisTemplate;
        this.key = key;
    }
    @Override
    public void run(){
        //以uuid为key，当前线程id为value保存到Redis中
        stringRedisTemplate.opsForValue().set(uuid, Thread.currentThread().getId());
         //定义更新锁的过期时间
        while(true){
            springRedisTemplate.expire(key, 30, TimeUnit.SECONDS);
            try{
                //每隔10秒执行一次
                Thread.sleep(10000);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```

接下来，我们定义一个ThreadUtils工具类，这个工具类中有一个根据线程id获取线程的方法getThreadByThreadId(long threadId)。

```
public class ThreadUtils{
    //根据线程id获取线程句柄
```

```
    public static Thread getThreadByThreadId(long threadId){
        ThreadGroup group = Thread.currentThread().getThreadGroup();
        while(group != null){
            Thread[] threads = new Thread[(int)(group.activeCount() * 1.2)];
            int count = group.enumerate(threads, true);
            for(int i = 0; i < count; i++){
                if(threadId == threads[i].getId()){
                    return threads[i];
                }
            }
        }
    }
}
```

上述解决分布式锁失效的问题在分布式锁领域有一个专业的术语叫做 **“异步续命”** 。需要注意的是：当业务代码执行完毕后，我们需要停止更新锁超时时间的线程。所以，这里，我对程序的改动是比较大的，首先，将更新锁超时的时间任务重新定义为一个UpdateLockTimeoutTask类，并将uuid和StringRedisTemplate注入到任务类中，在执行定时更新锁超时时间时，首先将当前线程保存到Redis中，其中Key为传递进来的uuid。

在首先获取分布式锁后，重新启动线程，并将uuid和StringRedisTemplate传递到任务类中执行任务。当业务代码执行完毕后，调用releaseLock()方法释放锁时，我们会通过uuid从Redis中获取更新锁超时时间的线程id，并通过线程id获取到更新锁超时时间的线程，调用线程的interrupt()方法来中断线程。

此时，当分布式锁释放后，更新锁超时的线程就会由于线程中断而退出了。

实现分布式锁的基本要求

结合上述的案例，我们可以得出实现分布式锁的基本要求：

- 支持互斥性
- 支持锁超时
- 支持阻塞和非阻塞特性
- 支持可重入性
- 支持高可用

通用分布式解决方案

在互联网行业，分布式锁是一个绕不开的话题，同时，也有很多通用的分布式锁解决方案，其中，用的比较多的一种方案就是使用开源的Redisson框架来解决分布式锁问题。

有关Redisson分布式锁的使用方案大家可以参考《【高并发】你知道吗？大家都在使用Redisson实现分布式锁了！！》

既然Redisson框架已经很牛逼了，我们直接使用Redisson框架是否能够100%的保证分布式锁不出问题呢？答案是无法100%的保证。因为在分布式领域没有哪一家公司或者架构师能够保证100%的不出问题，就连阿里这样的大公司、阿里的首席架构师这样的技术大牛也不敢保证100%的不出问题。

在分布式领域，无法做到100%无故障，我们追求的是几个9的目标，例如99.999%无故障。

CAP理论

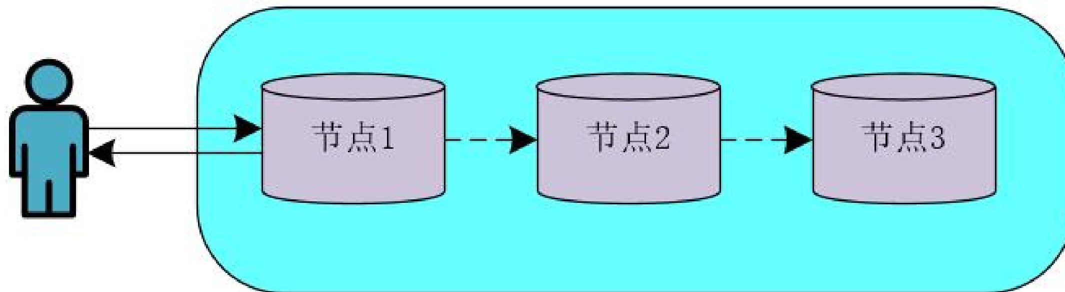
在分布式领域，有一个非常重要的理论叫做CAP理论。

- C：Consistency（一致性）
- A：Availability（可用性）
- P：Partition tolerance（分区容错性）

在分布式领域中，是必须要保证分区容错性的，也就是必须要保证“P”，所以，我们只能保证CP或者AP。

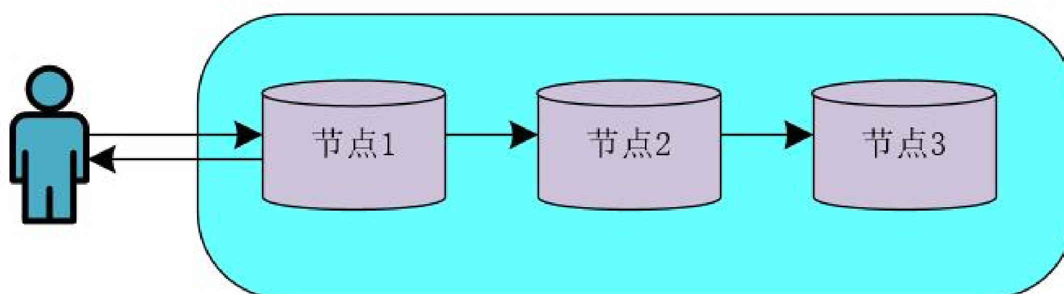
这里，我们可以使用Redis和Zookeeper来进行简单的对比，我们可以使用Redis实现AP架构的分布式锁，使用Zookeeper实现CP架构的分布式锁。

- 基于Redis的AP架构的分布式锁模型



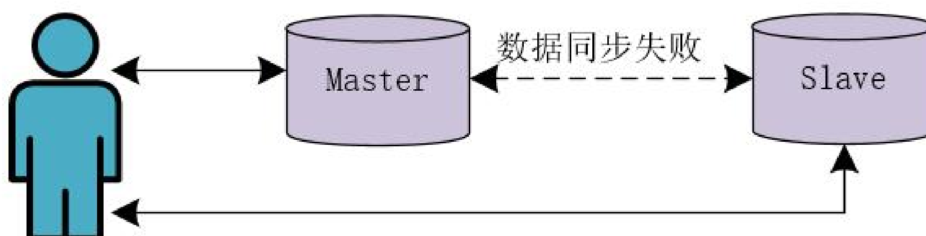
在基于Redis实现的AP架构的分布式锁模型中，向Redis节点1写入数据后，会立即返回结果，之后在Redis中会以异步的方式来同步数据。

- 基于Zookeeper的CP架构的分布式锁模型



在基于Zookeeper实现的CP架构的分布式模型中，向节点1写入数据后，会等待数据的同步结果，当数据在大多数Zookeeper节点间同步成功后，才会返回结果数据。

当我们使用基于Redis的AP架构实现分布式锁时，需要注意一个问题，这个问题可以使用下图来表示。



也就是Redis主从节点之间的数据同步失败，假设线程向Master节点写入了数据，而Redis中Master节点向Slave节点同步数据失败了。此时，另一个线程读取的Slave节点中的数据，发现没有添加分布式锁，此时就会出现问题了！！！

所以，**在设计分布式锁方案时，也需要注意Redis节点之间的数据同步问题。**

红锁的实现

在Redisson框架中，实现了红锁的机制，Redisson的RedissonRedLock对象实现了Redlock介绍的加锁算法。该对象也可以用来将多个RLock对象关联为一个红锁，每个RLock对象实例可以来自于不同的Redisson实例。当红锁中超过半数的RLock加锁成功后，才会认为加锁是成功的，这就提高了分布式锁的高可用。

我们可以使用Redisson框架来实现红锁。

```
public void testRedLock(RedissonClient redisson1,RedissonClient redisson2, RedissonClient redisson3){
    RLock lock1 = redisson1.getLock("lock1");
    RLock lock2 = redisson2.getLock("lock2");
    RLock lock3 = redisson3.getLock("lock3");
    RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);
    try {
        // 同时加锁：lock1 lock2 lock3, 红锁在大部分节点上加锁成功就算成功。
        lock.lock();
        // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁
        boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
```

其实，在实际场景中，红锁是很少使用的。这是因为使用了红锁后会影响高并发环境下的性能，使得程序的体验更差。所以，在实际场景中，我们一般都是要保证Redis集群的可靠性。同时，使用红锁后，当加锁成功的RLock个数不超过总数的一半时，会返回加锁失败，即使在业务层面任务加锁成功了，但是红锁也会返回加锁失败的结果。另外，使用红锁时，需要提供多套Redis的主从部署架构，同时，这多套Redis主从架构中的Master节点必须都是独立的，相互之间没有任何数据交互。

高并发“黑科技”与致胜奇招

假设，我们就是使用Redis来实现分布式锁，假设Redis的读写并发量在5万左右。我们的商城业务需要支持的并发量在100万左右。如果这100万的并发全部打入Redis中，Redis很可能就会挂掉，那么，我们如何解决这个问题呢？接下来，我们就一起来探讨这个问题。

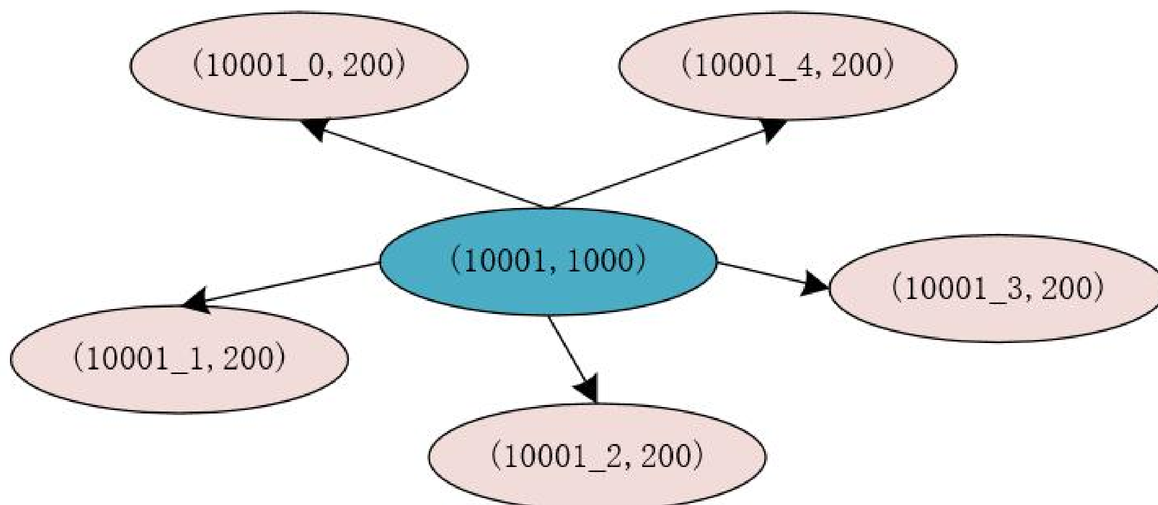
> 在高并发的商城系统中，如果采用Redis缓存数据，则Redis缓存的并发处理能力是关键，因为很多的前缀操作都需要访问Redis。而异步削峰只是基本的操作，关键还是要保证Redis的并发处理能力。

解决这个问题的关键思想就是：分而治之，将商品库存分开放。

暗度陈仓

我们在Redis中存储商品的库存数量时，可以将商品的库存进行“分割”存储来提升Redis的读写并发量。

例如，原来的商品的id为10001，库存为1000件，在Redis中的存储为(10001, 1000)，我们将原有的库存分割为5份，则每份的库存为200件，此时，我们在Redia中存储的信息为(10001_0, 200)，(10001_1, 200)，(10001_2, 200)，(10001_3, 200)，(10001_4, 200)。



此时，我们将库存进行分割后，每个分割后的库存使用商品id加上一个数字标识来存储，这样，在对存储商品库存的每个Key进行Hash运算时，得出的Hash结果是不同的，这就说明，存储商品库存的Key有很大概率不在Redis的同一个槽位中，这就能够提升Redis处理请求的性能和并发量。

分割库存后，我们还需要在Redis中存储一份商品id和分割库存后的Key的映射关系，此时映射关系的Key为商品的id，也就是10001，Value为分割库存后存储库存信息的Key，也就是10001_0，10001_1，10001_2，10001_3，10001_4。在Redis中我们可以使用List来存储这些值。

在真正处理库存信息时，我们可以先从Redis中查询出商品对应的分割库存后的所有Key，同时使用AtomicLong来记录当前的请求数量，使用请求数量对从Redia中查询出的商品对应的分割库存后的所有Key的长度进行求模运算，得出的结果为0，1，2，3，4。再在前面拼接上商品id就可以得出真正的库存缓存的Key。此时，就可以根据这个Key直接到Redis中获取相应的库存信息。

同时，我们可以将分隔的不同的库存数据分别存储到不同的Redis服务器中，进一步提升Redis的并发量。

移花接木

在高并发业务场景中，我们可以直接使用Lua脚本库（OpenResty）从负载均衡层直接访问缓存。

这里，我们思考一个场景：如果在高并发业务场景中，商品被瞬间抢购一空。此时，用户再发起请求时，如果系统由负载均衡层请求应用层的各个服务，再由应用层的各个服务访问缓存和数据库，其实，本质上已经没有任何意义了，因为商品已经卖完了，再通过系统的应用层进行层层校验已经没有太多意义了！！而应用层的并发访问量是以百为单位的，这又在一定程度上会降低系统的并发度。

为了解决这个问题，此时，**我们可以在系统的负载均衡层取出用户发送请求时携带的用户id，商品id和活动id等信息，直接通过Lua脚本等技术来访问缓存中的库存信息。如果商品的库存小于或者等于0，则直接返回用户商品已售完的提示信息，而不用再经过应用层的层层校验了。**

结束语

好了，《深入理解高并发编程（第1版）》到这儿就结束了，希望这本电子书能够给你带来实质性的帮助，我会持续更新【冰河技术】微信公众号的【高并发】专题文章。小伙伴们可以关注【冰河技术】微信公众号，第一时间阅读超硬核技术干货，我们一起进阶，一起牛逼！！

重磅福利

微信搜一搜【冰河技术】微信公众号，关注这个有深度的程序员，每天阅读超硬核技术干货，公众号内回复【PDF】有我准备的一线大厂面试资料和我原创的超硬核PDF技术文档，以及我为大家精心准备的多套简历模板（不断更新中），希望大家都能找到心仪的工作，学习是一条时而郁郁寡欢，时而开怀大笑的路，加油。如果你通过努力成功进入到了心仪的公司，一定不要懈怠放松，职场成长和新技术学习一样，不进则退。如果有幸我们江湖再见！

另外，我开源的各个PDF，后续我都会持续更新和维护，感谢大家长期以来对冰河的支持！！

写在最后

> 如果你觉得冰河写的还不错，请微信搜索并关注「 **冰河技术** 」微信公众号，跟冰河学习高并发、分布式、微服务、大数据、互联网和云原生技术，「 **冰河技术** 」微信公众号更新了大量技术专题，每一篇技术文章干货满满！不少读者已经通过阅读「 **冰河技术** 」微信公众号文章，吊打面试官，成功跳槽到大厂；也有不少读者实现了技术上的飞跃，成为公司的技术骨干！如果你也想像他们一样提升自己的能力，实现技术能力的飞跃，进大厂，升职加薪，那就关注「 **冰河技术** 」微信公众号吧，每天更新超硬核技术干货，让你对如何提升技术能力不再迷茫！