

# Java IO

## 同步、异步、阻塞、非阻塞

### 同步与异步

- **同步**: 同步就是发起一个调用后，被调用者未处理完请求之前，调用不返回。
- **异步**: 异步就是发起一个调用后，立刻得到被调用者的回应表示已接收到请求，但是被调用者并没有返回结果，此时我们可以处理其他的请求，被调用者通常依靠事件，回调等机制来通知调用者其返回结果。

同步和异步的区别最大在于异步的话调用者不需要等待处理结果，被调用者会通过回调等机制来通知调用者其返回结果。

### 阻塞和非阻塞

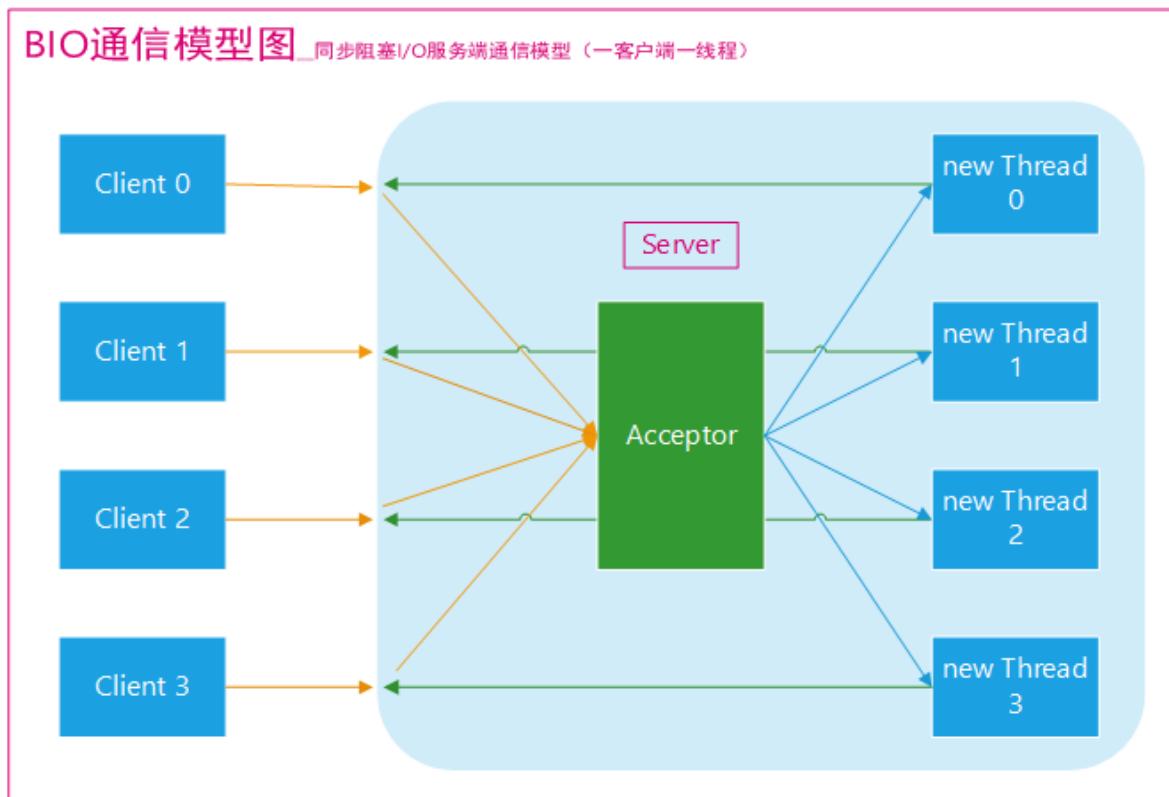
- **阻塞**: 阻塞就是发起一个请求，调用者一直等待请求结果返回，也就是当前线程会被挂起，无法从事其他任务，只有当条件就绪才能继续。
- **非阻塞**: 非阻塞就是发起一个请求，调用者不用一直等着结果返回，可以先去干其他事情。

## BIO (Blocking I/O)

同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。

### 传统 BIO

BIO通信（一请求一应答）模型图如下：



采用 **BIO 通信模型** 的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接。我们一般通过在 `while(true)` 循环中服务端会调用 `accept()` 方法等待接收客户端的连接的方式监听请求，请求一旦接收到一个连接请求，就可以建立通信套接字在这个通信套接字上进行读写操作，此时不能再接收其他客户端连接请求，只能等待同当前连接的客户端的操作执行完成，不过可以通过多线程来支持多个客户端的连接，如上图所示。

如果要让 **BIO 通信模型** 能够同时处理多个客户端请求，就必须使用多线程（主要原因是 `socket.accept()`、`socket.read()`、`socket.write()` 涉及的三个主要函数都是同步阻塞的），也就是说它在接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的 **一请求一应答通信模型**。

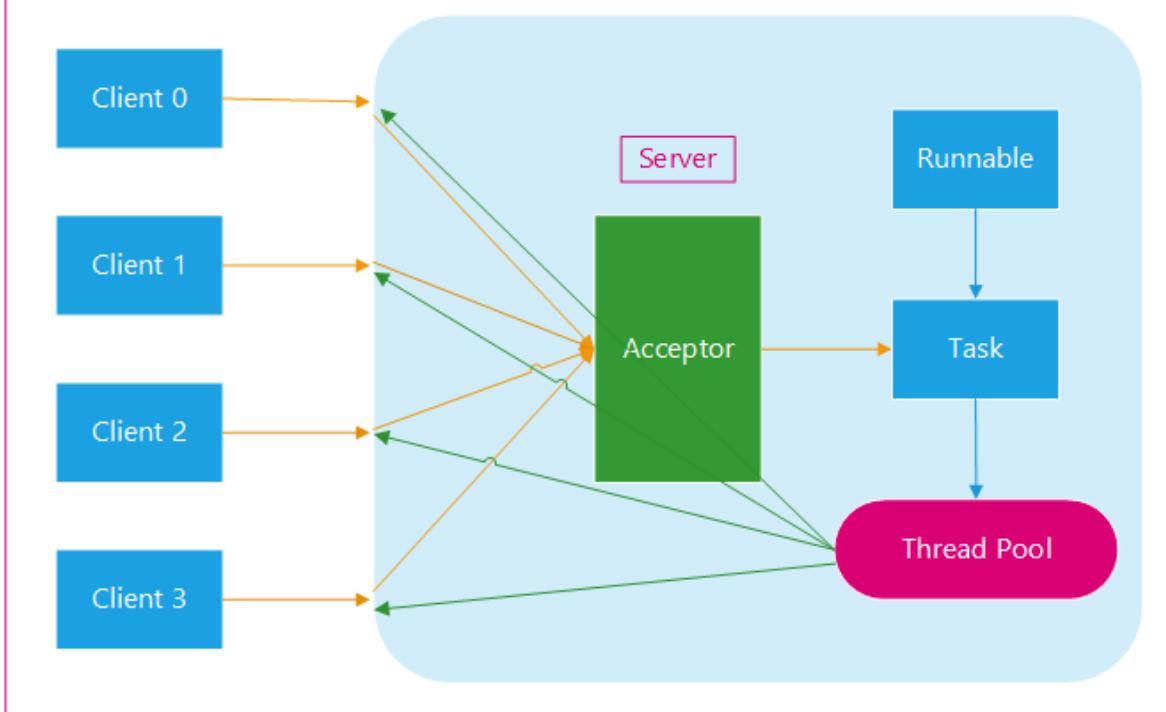
**我们再设想一下当客户端并发访问量增加后这种模型会出现什么问题？**

在 Java 虚拟机中，线程是宝贵的资源，线程的创建和销毁成本很高，除此之外，线程的切换成本也是很高的。尤其在 Linux 这样的操作系统中，线程本质上就是一个进程，创建和销毁线程都是重量级的系统函数。如果并发访问量增加会导致线程数急剧膨胀可能会导致线程堆栈溢出、创建新线程失败等问题，最终导致进程宕机或者僵死，不能对外提供服务。

## 伪异步 IO

为了解决同步阻塞I/O面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化——后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数M：线程池最大线程数N的比例关系，其中M可以远远大于N.通过线程池可以灵活地调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。

**伪异步IO模型图** 同步阻塞I/O服务端通信模型（N客户端M线程，N可以远远大于M）



采用线程池和任务队列可以实现一种叫做伪异步的 I/O 通信框架，它的模型图如上图所示。当有新的客户端接入时，将客户端的 Socket 封装成一个 Task (该任务实现 `java.lang.Runnable` 接口) 投递到后端的线程池中进行处理，JDK 的线程池维护一个消息队列和 N 个活跃线程，对消息队列中的任务进行处理。由于线程池可以设置消息队列的大小和最大线程数，因此，它的资源占用是可控的，无论多少个客户端并发访问，都不会导致资源的耗尽和宕机。

伪异步I/O通信框架采用了线程池实现，因此避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。不过因为它的底层仍然是同步阻塞的BIO模型，因此无法从根本上解决问题。

# 总结

在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的I/O并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的BIO模型是无能为力的。因此，我们需要一种更高效的I/O处理模型来应对更高的并发量。

## NIO (New I/O)

### NIO 简介

NIO是一种同步非阻塞的I/O模型，在Java 1.4中引入了NIO框架，对应java.nio包，提供了Channel，Selector，Buffer等抽象。

NIO中的N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。NIO提供了与传统BIO模型中的`Socket`和`ServerSocket`相对应的`SocketChannel`和`ServerSocketChannel`两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用NIO的非阻塞模式来开发。

### NIO的特性/NIO与IO区别

如果是在面试中回答这个问题，我觉得首先肯定要从NIO流是非阻塞IO而IO流是阻塞IO说起。然后，可以从NIO的3个核心组件/特性为NIO带来的一些改进来分析。如果，你把这些都回答上了我觉得你对于NIO就有了更为深入一点的认识，面试官问到你这个问题，你也能很轻松的回答上来了。

#### Non-blocking IO (非阻塞IO)

IO流是阻塞的，NIO流是不阻塞的。

Java NIO使我们可以进行非阻塞IO操作。比如说，单线程中从通道读取数据到buffer，同时可以继续做别的事情，当数据读取到buffer中后，线程再继续处理数据。写数据也是一样的。另外，非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。

Java IO的各种流是阻塞的。这意味着，当一个线程调用`read()`或`write()`时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了

#### Buffer(缓冲区)

IO面向流(Stream oriented)，而NIO面向缓冲区(Buffer oriented)。

Buffer是一个对象，它包含一些要写入或者要读出的数据。在NIO类库中加入Buffer对象，体现了新库与原I/O的一个重要区别。在面向流的I/O中，可以将数据直接写入或者将数据直接读到Stream对象中。虽然Stream中也有Buffer开头的扩展类，但只是流的包装类，还是从流读到缓冲区，而NIO却是直接读到Buffer中进行操作。

在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

最常用的缓冲区是ByteBuffer，一个 ByteBuffer 提供了一组功能用于操作 byte 数组。除了 ByteBuffer，还有其他的一些缓冲区，事实上，每一种 Java 基本类型（除了 Boolean 类型）都对应有一种缓冲区。

#### Channel (通道)

NIO 通过 Channel (通道) 进行读写。

通道是双向的，可读也可写，而流的读写是单向的。无论读写，通道只能和Buffer交互。因为 Buffer，通道可以异步地读写。

## Selectors(选择器)

NIO有选择器，而IO没有。

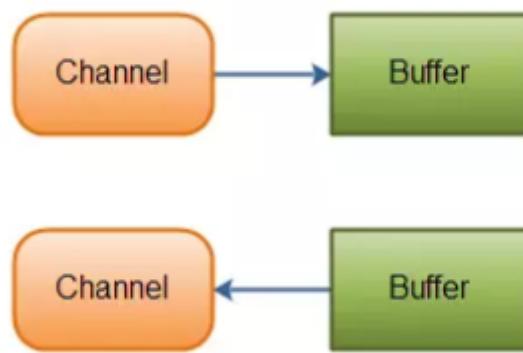
选择器用于使用单个线程处理多个通道。因此，它需要较少的线程来处理这些通道。线程之间的切换对于操作系统来说是昂贵的。因此，为了提高系统效率选择器是有用的。

## NIO 读数据和写数据方式

通常来说NIO中的所有IO都是从 Channel (通道) 开始的。

- 从通道进行数据读取：创建一个缓冲区，然后请求通道读取数据。
- 从通道进行数据写入：创建一个缓冲区，填充数据，并要求通道写入数据。

数据读取和写入操作图示：



## NIO核心组件简单介绍

NIO 包含下面几个核心的组件：

- Channel(通道)
- Buffer(缓冲区)
- Selector(选择器)

整个NIO体系包含的类远远不止这三个，只能说这三个是NIO体系的“核心API”。我们上面已经对这三个概念进行了基本的阐述，这里就不多做解释了。

为什么大家都愿意用 JDK 原生 NIO 进行开发呢？大家都可以看出来，是真的难用！除了编程复杂、编程模型难之外，它还有以下让人诟病的问题：

- JDK 的 NIO 底层由 epoll 实现，该实现饱受诟病的空轮询 bug 会导致 cpu 飙升 100% (Netty解决办法：记录select空转的次数，定义一个阈值，这个阈值默认是512，可以在应用层通过设置系统属性io.netty.selectorAutoRebuildThreshold传入，当空转的次数超过了这个阈值，重新构建新Selector，将老Selector上注册的Channel转移到新建的Selector上，关闭老Selector，用新的Selector代替老Selector，详细实现可以查看NioEventLoop中的selector和rebuildSelector方法)
- 项目庞大之后，自行实现的 NIO 很容易出现各类 bug，维护成本较高

Netty 的出现很大程度上改善了 JDK 原生 NIO 所存在的一些让人难以忍受的问题。

## AIO (Asynchronous I/O)

AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。（除了 AIO 其他的 IO 类型都是同步的，这一点可以从底层IO线程模型解释，推荐一篇文章：[《漫话：如何给女朋友解释什么是Linux的五种IO模型？》](#)）

查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

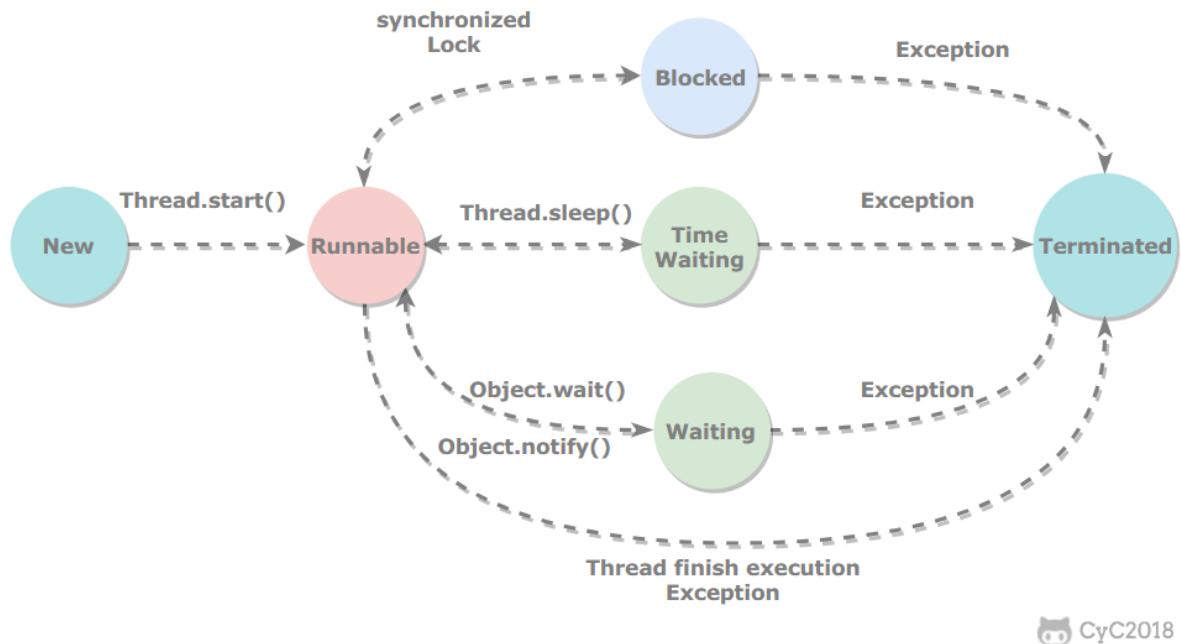
## BIO,NIO,AIO 有什么区别？

---

- **BIO (Blocking I/O):** 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (New I/O):** NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector , Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

# Java并发

## 线程状态转换



CyC2018

## sleep和wait的区别

- sleep是Thread类的方法，wait是Object类中定义的方法
- sleep方法可以在任何地方使用
- wait只能在synchronized方法或synchronized块中使用
- sleep只会让出CPU，不会导致锁行为的改变；wait不仅让出CPU，还会释放已经占有的同步资源锁

## 创建线程

有三种使用线程的方法：

- 实现 Runnable 接口；
- 实现 Callable 接口；
- 继承 Thread 类。

实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。可以说任务是通过线程驱动从而执行的。

### 实现 Runnable 接口

需要实现 run() 方法。

通过 Thread 调用 start() 方法来启动线程。

```
public class MyRunnable implements Runnable {  
    public void run() {  
        // ...  
    }  
    public static void main(String[] args) {  
        MyRunnable instance = new MyRunnable();  
        Thread thread = new Thread(instance);  
        thread.start();  
    }  
}
```

## 实现 Callable 接口

与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

```
public class MyCallable implements Callable<Integer> {  
    public Integer call() {  
        return 123;  
    }  
    public static void main(String[] args) throws ExecutionException,  
    InterruptedException {  
        MyCallable mc = new MyCallable();  
        FutureTask<Integer> ft = new FutureTask<>(mc);  
        Thread thread = new Thread(ft);  
        thread.start();  
        System.out.println(ft.get());  
    }  
}
```

## 继承 Thread 类

同样也是需要实现 run() 方法，因为 Thread 类也实现了 Runnable 接口。

当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。

```
public class MyThread extends Thread {  
    public void run() {  
        // ...  
    }  
    public static void main(String[] args) {  
        MyThread mt = new MyThread();  
        mt.start();  
    }  
}
```

## Daemon

守护线程是程序运行时在后台提供服务的线程（如垃圾回收线程），不属于程序中不可或缺的部分。

当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。

main() 属于非守护线程。

使用 setDaemon() 方法将一个线程设置为守护线程。

```
public static void main(String[] args) {
    Thread thread = new Thread(new MyRunnable());
    thread.setDaemon(true);
}
```

## 线程池

### 研读ThreadPoolExecutor

看一下该类的构造器：

```
public ThreadPoolExecutor(int paramInt1, int paramInt2, long paramLong, TimeUnit
paramTimeUnit, BlockingQueue<Runnable> paramBlockingQueue, ThreadFactory
paramThreadFactory, RejectedExecutionHandler paramRejectedExecutionHandler) {
    this.ctl = new AtomicInteger(ctlOf(-536870912, 0));
    this.mainLock = new ReentrantLock();
    this.workers = new HashSet();
    this.termination = this.mainLock.newCondition();
    if ((paramInt1 < 0) || (paramInt2 <= 0) || (paramInt2 < paramInt1) ||
(paramLong < 0L))
        throw new IllegalArgumentException();
    if ((paramBlockingQueue == null) || (paramThreadFactory == null) ||
(paramRejectedExecutionHandler == null))
        throw new NullPointerException();
    this.corePoolSize = paramInt1;
    this.maximumPoolSize = paramInt2;
    this.workQueue = paramBlockingQueue;
    this.keepAliveTime = paramTimeUnit.toNanos(paramLong);
    this.threadFactory = paramThreadFactory;
    this.handler = paramRejectedExecutionHandler;
}
```

**corePoolSize**: 线程池的核心池大小，在创建线程池之后，线程池默认没有任何线程。

当有任务过来的时候才会去创建线程执行任务。换个说法，线程池创建之后，线程池中的线程数为0，当任务过来就会创建一个线程去执行，直到线程数达到corePoolSize 之后，就会被到达的任务放在队列中。（注意是到达的任务）。换句更精炼的话：corePoolSize 表示线程池中允许同时运行的最大线程数。即便是线程池里没有任何任务，也会有corePoolSize个线程在候着等任务。

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

**maximumPoolSize** : 线程池允许的最大线程数，他表示最大能创建多少个线程。maximumPoolSize 肯定是大于等于corePoolSize。

**keepAliveTime** :表示线程没有任务时最多保持多久然后停止。默认情况下，只有线程池中线程数大于corePoolSize 时，keepAliveTime 才会起作用。换句话说，当线程池中的线程数大于corePoolSize，并且一个线程空闲时间达到了keepAliveTime，那么就是shutdown。

**Unit**: keepAliveTime 的单位。

**workQueue**: 一个阻塞队列，用来存储等待执行的任务，当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能。

**threadFactory**: 线程工厂，用来创建线程。

**handler**: 表示当拒绝处理任务时的策略。

## 任务缓存队列

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

workQueue的类型为BlockingQueue，通常可以取下面三种类型：

- 1) 有界任务队列ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；
- 2) 无界任务队列LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；
- 3) 直接提交队列synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

## 拒绝策略

AbortPolicy：丢弃任务并抛出RejectedExecutionException

CallerRunsPolicy：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。

DiscardOldestPolicy：丢弃队列中最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。

DiscardPolicy：丢弃任务，不做任何处理。

## 线程池的任务处理策略：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目 $\geq$ corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

## 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等**所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务**

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

## 常见的四种线程池

### newFixedThreadPool

```

public static ExecutorService newFixedThreadPool(int var0) {
    return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue());
}

public static ExecutorService newFixedThreadPool(int var0, ThreadFactory
var1) {
    return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue(), var1);
}

```

固定大小的线程池，可以指定线程池的大小，该线程池corePoolSize和maximumPoolSize相等，阻塞队列使用的是LinkedBlockingQueue，大小为整数最大值。

该线程池中的线程数量始终不变，当有新任务提交时，线程池中有空闲线程则会立即执行，如果没有，则会暂存到阻塞队列。对于固定大小的线程池，不存在线程数量的变化。同时使用无界的LinkedBlockingQueue来存放执行的任务。当任务提交十分频繁的时候，LinkedBlockingQueue迅速增大，存在着耗尽系统资源的问题。而且在线程池空闲时，即线程池中没有可运行任务时，它也不会释放工作线程，还会占用一定的系统资源，需要shutdown。

### **newSingleThreadExecutor**

```

public static ExecutorService newSingleThreadExecutor() {
    return new Executors.FinalizableDelegatedExecutorService(
        new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue()));
}

public static ExecutorService newSingleThreadExecutor(ThreadFactory var0) {
    return new Executors.FinalizableDelegatedExecutorService(
        new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue(), var0));
}

```

单个线程线程池，只有一个线程的线程池，阻塞队列使用的是LinkedBlockingQueue,若有多余的任务提交到线程池中，则会被暂存到阻塞队列，待空闲时再去执行。按照先入先出的顺序执行任务。

### **newCachedThreadPool**

```

public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new
SynchronousQueue());
}

public static ExecutorService newCachedThreadPool(ThreadFactory var0) {
    return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new
SynchronousQueue(), var0);
}

```

缓存线程池，缓存的线程默认存活60秒。线程的核心池corePoolSize大小为0，核心池最大为Integer.MAX\_VALUE,阻塞队列使用的是SynchronousQueue。是一个直接提交的阻塞队列，他总会迫使线程池增加新的线程去执行新的任务。在没有任务执行时，当线程的空闲时间超过keepAliveTime（60秒），则工作线程将会终止被回收，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销。如果同时有大量任务被提交，而且任务执行的时间不是特别快，那么线程池便会新增出等量的线程池处理任务，这很可能会很快耗尽系统的资源。

## newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int var0) {  
    return new ScheduledThreadPoolExecutor(var0);  
}  
  
public static ScheduledExecutorService newScheduledThreadPool(int var0,  
ThreadFactory var1) {  
    return new ScheduledThreadPoolExecutor(var0, var1);  
}
```

定时线程池，该线程池可用于周期性地去执行任务，通常用于周期性的同步数据。

scheduleAtFixedRate:是以固定的频率去执行任务，周期是指每次执行任务成功执行之间的间隔。

schedultWithFixedDelay:是以固定的延时去执行任务，延时是指上一次执行成功之后和下一次开始执行的之前的时间。

## 如何选择线程池线程数量

线程池的大小决定着系统的性能，过大或者过小的线程池数量都无法发挥最优的系统性能。

当然线程池的大小也不需要做的太过于精确，只需要避免过大和过小的情况。一般来说，确定线程池的大小需要考虑CPU的数量，内存大小，任务是计算密集型还是IO密集型等因素

**NCPU = CPU的数量**

**UCPU = 期望对CPU的使用率  $0 \leq UCPU \leq 1$**

**W/C = 等待时间与计算时间的比率**

如果希望处理器达到理想的使用率，那么线程池的最优大小为：

线程池大小= $NCPU * UCPU(1+W/C)$

## 控制线程执行的顺序

### 方法一：join

```
public class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread t1 = new Thread(new MyThread1());  
        Thread t2 = new Thread(new MyThread2());  
        Thread t3 = new Thread(new MyThread3());  
        t1.start();  
        t1.join();  
        t2.start();  
        t2.join();  
        t3.start();  
    }  
}  
  
class MyThread1 implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("I am thread 1");  
    }  
}
```

```

class MyThread2 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 2");
    }
}

class MyThread3 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 3");
    }
}

```

join方法：让主线程等待子线程运行结束后再继续运行

有了join方法的帮助，线程123就能按照指定的顺序执行了。

我们来看看示例当中主线程与子线程的执行顺序。在main方法中，先是调用了t1.start方法，启动t1线程，随后调用t1的join方法，main所在的主线程就需要等待t1子线程中的run方法运行完成后才能继续运行，所以主线程卡在t2.start方法之前等待t1线程。等t1运行完后，主线程重新获得主动权，继续运行t2.start和t2.join方法，与t1子线程类似，main主线程等待t2完成后继续执行，如此执行下去，join方法就有效的解决了执行顺序问题。因为在同一个时间点，各个线程是同步状态。

当然解决方法不止一个：

## 方法二：Executors.newSingleThreadExecutor()

```

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

public class Test {
    private static ExecutorService executor =
    Executors.newSingleThreadExecutor();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());
        Thread t3 = new Thread(new MyThread3());
        executor.submit(t1);
        executor.submit(t2);
        executor.submit(t3);
        executor.shutdown();
    }
}

class MyThread1 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 1");
    }
}

class MyThread2 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 2");
    }
}

```

```

}

class MyThread3 implements Runnable {
    @Override
    public void run() {
        System.out.println("I am thread 3");
    }
}

```

利用并发包里的Excutors的newSingleThreadExecutor产生一个单线程的线程池，而这个线程池的底层原理就是一个先进先出（FIFO）的队列。代码中executor.submit依次添加了123线程，按照FIFO的特性，执行顺序也就是123的执行结果，从而保证了执行顺序。

### 方法三：wait()和notify()

```

public class QueueThread implements Runnable{
    private Object current;
    private Object next;
    private int max=100;
    private String word;
    public QueueThread(Object current, Object next, String word) {
        this.current = current;
        this.next = next;
        this.word = word;
    }
    @Override
    public void run() {
        // TODO Auto-generated method stub
        for(int i=0;i<max;i++){
            synchronized (current) {
                synchronized (next) {
                    System.out.println(word);
                    next.notify();
                }
                try {
                    current.wait();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
        //必须做一下这样处理，否则thread1-thread4停不了
        synchronized (next) {
            next.notify();
            System.out.println(Thread.currentThread().getName()+"执行完毕");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        Object a = new Object();
        Object b = new Object();
        Object c = new Object();
        Object d = new Object();
        Object e = new Object();
        //之所以每次当前线程都要sleep(10)是为了保证线程的执行顺序
    }
}

```

```

        new Thread(new QueueThread(a,b,"a")).start();
        Thread.sleep(10);
        new Thread(new QueueThread(b,c,"b")).start();
        Thread.sleep(10);
        new Thread(new QueueThread(c,d,"c")).start();
        Thread.sleep(10);
        new Thread(new QueueThread(d,e,"d")).start();
        Thread.sleep(10);
        Thread thread4 = new Thread(new QueueThread(e,a,"e"));
        thread4.start();
        thread4.join();
        //因为线程0-4停止是依次执行的，所以如果保证主线程在线程4后停止，那么就能保证主线程是最
        //后关闭的
        System.out.println("程序耗时：" + (System.currentTimeMillis()-startTime));
    }
}

```

首先，我们保证了线程0-线程4依次启动，并设置了Thread.sleep(10)，保证线程0-4依次执行他们的run方法。

其次，我们看QueueThread的run()便可知：1.线程获得current锁，2.获得next锁。3.打印并notify拥有next锁的一个对象4.线程执行current.wait(),释放current锁对象，并使线程处于阻塞状态。

然后，假设已经执行到了thread-4的run方法，那么此时的情况是这样的：

线程0处于阻塞状态，需要a.notify()才能使其回到runnable状态

线程1处于阻塞状态，需要b.notify()才能使其回到runnable状态

线程2处于阻塞状态，需要c.notify()才能使其回到runnable状态

线程3处于阻塞状态，需要d.notify()才能使其回到runnable状态

而线程4恰好可以需要执行a.notify()，所以能够使线程0回到runnable状态。然后执行e.wait()方法，使自身线程阻塞，需要e.notify()才能唤醒。

## 互斥同步（synchronized和ReentrantLock）

Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问，第一个是 JVM 实现的 synchronized，而另一个是 JDK 实现的 ReentrantLock。

### synchronized

#### 1. 同步一个代码块

```

public void func() {
    synchronized (this) {
        // ...
    }
}

```

它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步。

对于以下代码，使用 ExecutorService 执行了两个线程，由于调用的是同一个对象的同步代码块，因此这两个线程会进行同步，当一个线程进入同步语句块时，另一个线程就必须等待。

```

public class SynchronizedExample {

```

```
public void func1() {
    synchronized (this) {
        for (int i = 0; i < 10; i++) {
            System.out.print(i + " ");
        }
    }
}

public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func1());
    executorService.execute(() -> e1.func1());
}
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

对于以下代码，两个线程调用了不同对象的同步代码块，因此这两个线程就不需要同步。从输出结果可以看出，两个线程交叉执行。

```
public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    SynchronizedExample e2 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func1());
    executorService.execute(() -> e2.func1());
}
```

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

## 2. 同步一个方法

```
public synchronized void func () {
    // ...
}
```

它和同步代码块一样，作用于同一个对象。

## 3. 同步一个类

```
public void func() {
    synchronized (SynchronizedExample.class) {
        // ...
    }
}
```

作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步。

```
public class SynchronizedExample {

    public void func2() {
        synchronized (SynchronizedExample.class) {
            for (int i = 0; i < 10; i++) {
                System.out.print(i + " ");
            }
        }
    }
}
```

```

        }
    }
}

public static void main(String[] args) {
    SynchronizedExample e1 = new SynchronizedExample();
    SynchronizedExample e2 = new SynchronizedExample();
    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> e1.func2());
    executorService.execute(() -> e2.func2());
}

```

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

#### 4. 同步一个静态方法

```

public synchronized static void fun() {
    // ...
}

```

作用于整个类。

## ReentrantLock

ReentrantLock 是 java.util.concurrent (J.U.C) 包中的锁。

```

public class LockExample {

    private Lock lock = new ReentrantLock();

    public void func() {
        lock.lock();
        try {
            for (int i = 0; i < 10; i++) {
                System.out.print(i + " ");
            }
        } finally {
            lock.unlock(); // 确保释放锁，从而避免发生死锁。
        }
    }

    public static void main(String[] args) {
        LockExample lockExample = new LockExample();
        ExecutorService executorService = Executors.newCachedThreadPool();
        executorService.execute(() -> lockExample.func());
        executorService.execute(() -> lockExample.func());
    }
}

```

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

## 比较

### 1. 锁的实现

synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。

## 2. 性能

新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等，synchronized 与 ReentrantLock 大致相同。

## 3. 等待可中断

当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。

ReentrantLock 可中断，而 synchronized 不行。

## 4. 公平锁

公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。

synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的。

## 5. 锁绑定多个条件

一个 ReentrantLock 可以同时绑定多个 Condition 对象。

# 使用选择

除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized。这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放。

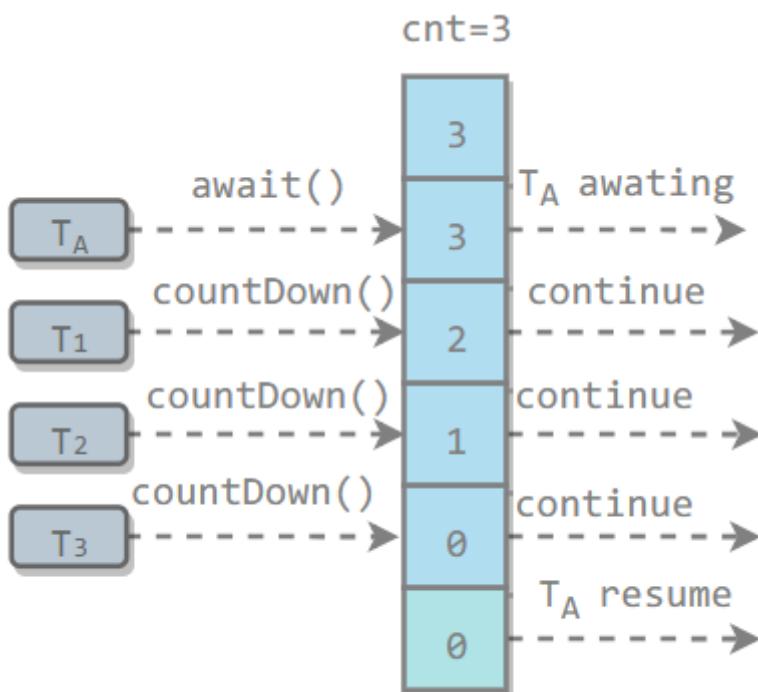
# J.U.C-AQS

java.util.concurrent (J.U.C) 大大提高了并发性能，AQS 被认为是 J.U.C 的核心。

## CountDownLatch

用来控制一个线程等待多个线程。

维护了一个计数器 cnt，每次调用 countDown() 方法会让计数器的值减 1，减到 0 的时候，那些因为调用 await() 方法而在等待的线程就会被唤醒。



```
public class CountdownLatchExample {  
  
    public static void main(String[] args) throws InterruptedException {  
        final int totalThread = 10;  
        CountDownLatch countDownLatch = new CountDownLatch(totalThread);  
        ExecutorService executorService = Executors.newCachedThreadPool();  
        for (int i = 0; i < totalThread; i++) {  
            executorService.execute(() -> {  
                System.out.print("run..");  
                countDownLatch.countDown();  
            });  
        }  
        countDownLatch.await();  
        System.out.println("end");  
        executorService.shutdown();  
    }  
}
```

```
run..run..run..run..run..run..run..run..run..end
```

## CyclicBarrier

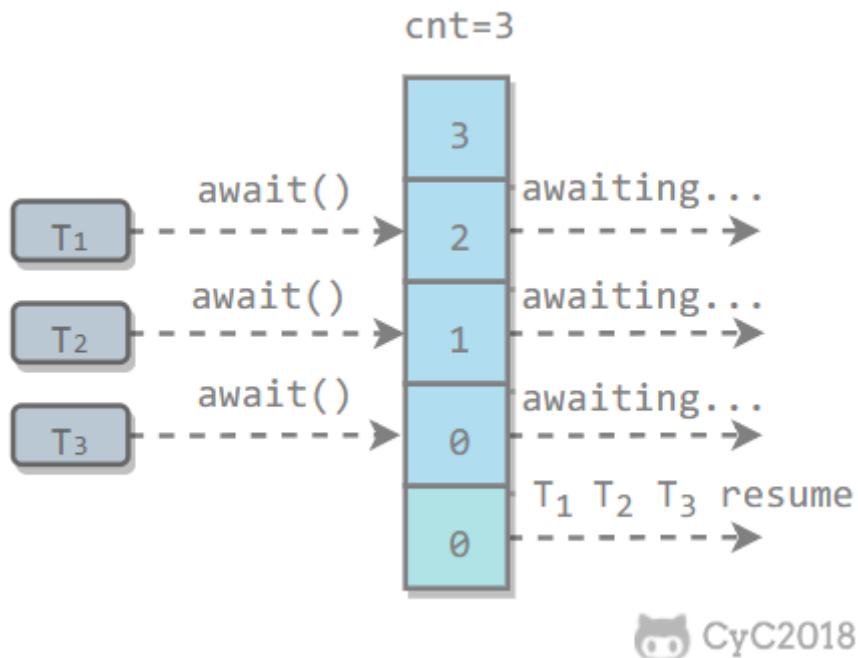
用来控制多个线程互相等待，只有当多个线程都到达时，这些线程才会继续执行。

和 CountdownLatch 相似，都是通过维护计数器来实现的。线程执行 await() 方法之后计数器会减 1，并进行等待，直到计数器为 0，所有调用 await() 方法而在等待的线程才能继续执行。

CyclicBarrier 和 CountdownLatch 的一个区别是，CyclicBarrier 的计数器通过调用 reset() 方法可以循环使用，所以它才叫做循环屏障。

CyclicBarrier 有两个构造函数，其中 parties 指示计数器的初始值，barrierAction 在所有线程都到达屏障的时候会执行一次。

```
public CyclicBarrier(int parties, Runnable barrierAction) {  
    if (parties <= 0) throw new IllegalArgumentException();  
    this.parties = parties;  
    this.count = parties;  
    this.barrierCommand = barrierAction;  
}  
  
public CyclicBarrier(int parties) {  
    this(parties, null);  
}
```



CyC2018

```
public class CyclicBarrierExample {

    public static void main(String[] args) {
        final int totalThread = 10;
        CyclicBarrier cyclicBarrier = new CyclicBarrier(totalThread);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < totalThread; i++) {
            executorService.execute(() -> {
                System.out.print("before..");
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException | BrokenBarrierException e) {
                    e.printStackTrace();
                }
                System.out.print("after..");
            });
        }
        executorService.shutdown();
    }
}
```

before..before..before..before..before..before..before..before..before..  
after..after..after..after..after..after..after..after..after..

## Semaphore

Semaphore 类似于操作系统中的信号量，可以控制对互斥资源的访问线程数。

以下代码模拟了对某个服务的并发请求，每次只能有 3 个客户端同时访问，请求总数为 10。

```
public class SemaphoreExample {

    public static void main(String[] args) {
        final int clientCount = 3;
        final int totalRequestCount = 10;
        Semaphore semaphore = new Semaphore(clientCount);
```

```

ExecutorService executorService = Executors.newCachedThreadPool();
for (int i = 0; i < totalRequestCount; i++) {
    executorService.execute(()->{
        try {
            semaphore.acquire();
            System.out.print(semaphore.availablePermits() + " ");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
        }
    });
}
executorService.shutdown();
}
}

```

2 1 2 2 2 2 2 1 2 2

## 使用 BlockingQueue 实现生产者消费者问题

```

public class ProducerConsumer {

    private static BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);

    private static class Producer extends Thread {
        @Override
        public void run() {
            try {
                queue.put("product");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.print("produce..");
        }
    }

    private static class Consumer extends Thread {

        @Override
        public void run() {
            try {
                String product = queue.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.print("consume..");
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 2; i++) {
            Producer producer = new Producer();
            producer.start();
        }
    }
}

```

```
for (int i = 0; i < 5; i++) {
    Consumer consumer = new Consumer();
    consumer.start();
}
for (int i = 0; i < 3; i++) {
    Producer producer = new Producer();
    producer.start();
}
}
```

```
produce..produce..consume..consume..produce..consume..produce..consume..produce..
.consume..
```

## 非阻塞同步

互斥同步最主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步。

互斥同步属于一种悲观的并发策略，总是认为只要不去做正确的同步措施，那就肯定会出现问题。无论共享数据是否真的会出现竞争，它都要进行加锁（这里讨论的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁）、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作。

### 1. CAS

随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略：先进行操作，如果没有其它线程争用共享数据，那操作就成功了，否则采取补偿措施（不断地重试，直到成功为止）。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步。

乐观锁需要操作和冲突检测这两个步骤具备原子性，这里就不能再使用互斥同步来保证了，只能靠硬件来完成。硬件支持的原子性操作最典型的是：比较并交换（Compare-and-Swap，CAS）。CAS 指令需要有 3 个操作数，分别是内存地址 V、旧的预期值 A 和新值 B。当执行操作时，只有当 V 的值等于 A，才将 V 的值更新为 B。

### 2. AtomicInteger

J.U.C 包里面的整数原子类 AtomicInteger 的方法调用了 Unsafe 类的 CAS 操作。

以下代码使用了 AtomicInteger 执行了自增的操作。

```
private AtomicInteger cnt = new AtomicInteger();

public void add() {
    cnt.incrementAndGet();
}
```

以下代码是 incrementAndGet() 的源码，它调用了 Unsafe 的 getAndAddInt()。

```
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

以下代码是 getAndAddInt() 源码，var1 指示对象内存地址，var2 指示该字段相对对象内存地址的偏移，var4 指示操作需要加的数值，这里为 1。通过 getIntVolatile(var1, var2) 得到旧的预期值，通过调用 compareAndSwapInt() 来进行 CAS 比较，如果该字段内存地址中的值等于 var5，那么就更新内存地址为 var1+var2 的变量为 var5+var4。

可以看到 getAndAddInt() 在一个循环中进行，发生冲突的做法是不断的进行重试。

```
public final int getAndAddInt(Object var1, long var2, int var4) {  
    int var5;  
    do {  
        var5 = this.getIntVolatile(var1, var2);  
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));  
  
    return var5;  
}
```

### 3. ABA

如果一个变量初次读取的时候是 A 值，它的值被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过。

J.U.C 包提供了一个带有标记的原子引用类 AtomicStampedReference 来解决这个问题，它可以通过控制变量值的版本来保证 CAS 的正确性。大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会比原子类更高效。

## JDK1.6 之后的底层优化

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

### ①偏向锁

引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步！关于偏向锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

### ②轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段（1.6之后加入的）。轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了CAS操作。关于轻量级锁的加锁和解锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

#### ③ 自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。

百度百科对自旋锁的解释：

何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过 `--xx:+UseSpinning` 参数来开启。JDK1.6 及 1.6 之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。**自旋次数的默认值是10次，用户可以修改--XX:PreBlockSpin来更改。**

另外，在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。

#### ④ 锁消除

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

#### ⑤ 锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小，一直在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

## synchronized底层

**synchronized 关键字底层原理属于 JVM 层面。**

Monitor：每个Java对象天生自带了一把看不见的锁

#### ① synchronized 同步语句块的情况

```

public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
            System.out.println("synchronized 代码块");
        }
    }
}

```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 `javac SynchronizedDemo.java` 命令生成编译后的 .class 文件，然后执行 `javap -c -s -v -l SynchronizedDemo.class`。

```

public void method();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=3, args_size=1
  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
Exception table:
  from   to target type
    4    14    17  any
   17    20    17  any
LineNumberTable:
  line 5: 0
  line 6: 4
  line 7: 12
  line 8: 22
StackMapTable: number_of_entries = 2
  frame_type = 255 /* full_frame */
  offset_delta = 17
  locals = [ class test/SynchronizedDemo, class java/lang/Object ]
  stack = [ class java/lang/Throwable ]
  frame_type = 250 /* chop */
  offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"

```

从上面我们可以看出：

**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置， monitorexit 指令则指明同步代码块的结束位置。**  
当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor 对象存在于每个 Java 对象的对象头中， synchronized 锁便是通过这种方式获取锁的，也是为什么 Java 中任意对象可以作为锁的原因) 的所有权。当计数器为 0 则可以成功获取，获取后将锁计数器设为 1 也就是加 1。相应的在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

## ② synchronized 修饰方法的情况

```

public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("synchronized 方法");
    }
}

```

```

public test.SynchronizedDemo2();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
  0: aload_0
  1: invokespecial #1           // Method java/lang/Object."<init>":()V
  4: return
LineNumberTable:
  line 3: 0

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 synchronized 鐧规礎
  5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 5: 0
  line 6: 8
}
SourceFile: "SynchronizedDemo2.java"

```

synchronized 修饰的方法并没有 monitoreenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

## ThreadLocal

### 1. ThreadLocal简介

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。**如果想实现每一个线程都有自己的专属本地变量该如何解决呢？** JDK 中提供的 ThreadLocal 类正是为了解决这样的问题。ThreadLocal 类主要解决的就是让每个线程绑定自己的值，可以将 ThreadLocal 类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

**如果你创建了一个 ThreadLocal 变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是 ThreadLocal 变量名的由来。他们可以使用 get() 和 set() 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。**

### 2. ThreadLocal示例

先通过下面这个实例来理解 ThreadLocal 的用法。先声明一个 ThreadLocal 对象，存储布尔类型的数值。然后分别在 main 线程、Thread1、Thread2 中为 ThreadLocal 对象设置不同的数值：

```

public class ThreadLocalDemo {
    public static void main(String[] args) {

        // 声明 ThreadLocal对象
        ThreadLocal<Boolean> mThreadLocal = new ThreadLocal<Boolean>();

        // 在主线程、子线程1、子线程2中去设置访问它的值
        mThreadLocal.set(true);
    }
}

```

```

        System.out.println("Main " + mThreadLocal.get());

        new Thread("Thread#1"){
            @Override
            public void run() {
                mThreadLocal.set(false);
                System.out.println("Thread#1 " + mThreadLocal.get());
            }
        }.start();

        new Thread("Thread#2"){
            @Override
            public void run() {
                System.out.println("Thread#2 " + mThreadLocal.get());
            }
        }.start();
    }
}

```

打印的结果输出如下所示：

```

MainThread true
Thread#1 false
Thread#2 null

```

可以看见，在不同线程对同一个 ThreadLocal 对象设置数值，在不同的线程中取出来的值不一样。接下来就分析一下源码，看看其内部结构。

### 3. ThreadLocal 原理

从 `Thread` 类源代码入手。

```

public class Thread implements Runnable {
    .....
    //与此线程有关的ThreadLocal值。由ThreadLocal类维护
    ThreadLocal.ThreadLocalMap threadLocals = null;

    //与此线程有关的InheritableThreadLocal值。由InheritableThreadLocal类维护
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
    .....
}

```

从上面 `Thread` 类源代码可以看出 `Thread` 类中有一个 `threadLocals` 和一个 `inheritableThreadLocals` 变量，它们都是 `ThreadLocalMap` 类型的变量，我们可以把 `ThreadLocalMap` 理解为 `ThreadLocal` 类实现的定制化的 `HashMap`。默认情况下这两个变量都是 `null`，只有当前线程调用 `ThreadLocal` 类的 `set` 或 `get` 方法时才创建它们，实际上调用这两个方法的时候，我们调用的是 `ThreadLocalMap` 类对应的 `get()`、`set()` 方法。

`ThreadLocal` 类的 `set()` 方法

```

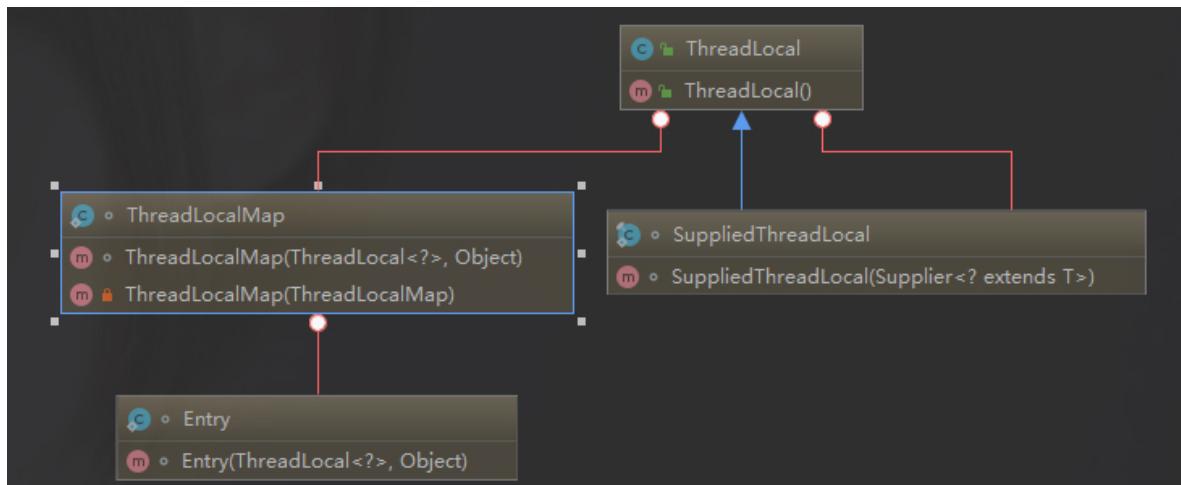
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

```

通过上面这些内容，我们足以通过猜测得出结论：最终的变量是放在了当前线程的 ThreadLocalMap 中，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是 ThreadLocalMap 的封装，传递了变量值。

每个 Thread 中都具备一个 ThreadLocalMap，而 ThreadLocalMap 可以存储以 ThreadLocal 为 key 的键值对。这里解释了为什么每个线程访问同一个 ThreadLocal，得到的确是不同的数值。另外，ThreadLocal 是 map 结构是为了让每个线程可以关联多个 ThreadLocal 变量。

ThreadLocalMap 是 ThreadLocal 的静态内部类。



## 4. ThreadLocal 内存泄露问题

ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候会 key 会被清理掉，而 value 不会被清理掉。这样一来，ThreadLocalMap 中就会出现 key 为 null 的 Entry。假如我们不做任何措施的话，value 永远无法被 GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap 实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。使用完 ThreadLocal 方法后最好手动调用 remove() 方法。

```

static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}

```

## 弱引用介绍：

如果一个对象只具有弱引用，那就类似于**可有可无的生活用品**。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

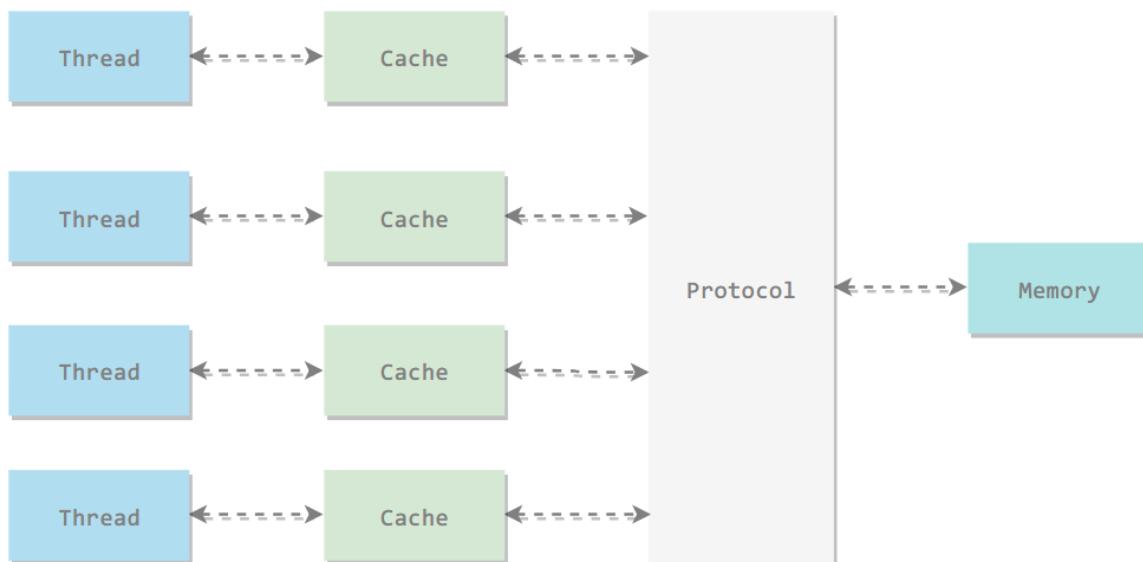
# Java 内存模型

Java内存模型试图屏蔽各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。

## 主内存与工作内存

处理器上的寄存器的读写的速度比内存快几个数量级，为了解决这种速度矛盾，在它们之间加入了高速缓存。

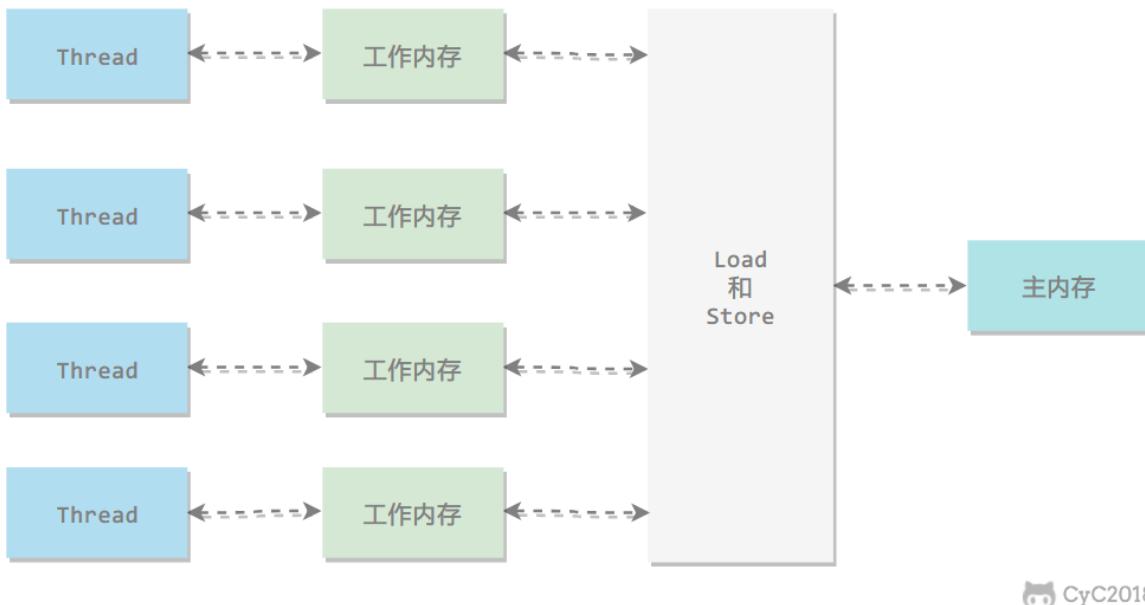
加入高速缓存带来了一个新的问题：缓存一致性。如果多个缓存共享同一块主内存区域，那么多个缓存的数据可能会不一致，需要一些协议来解决这个问题。



 CyC2018

所有的变量都存储在主内存中，每个线程还有自己的工作内存，工作内存存储在高速缓存或者寄存器中，保存了该线程使用的变量的主内存副本拷贝。

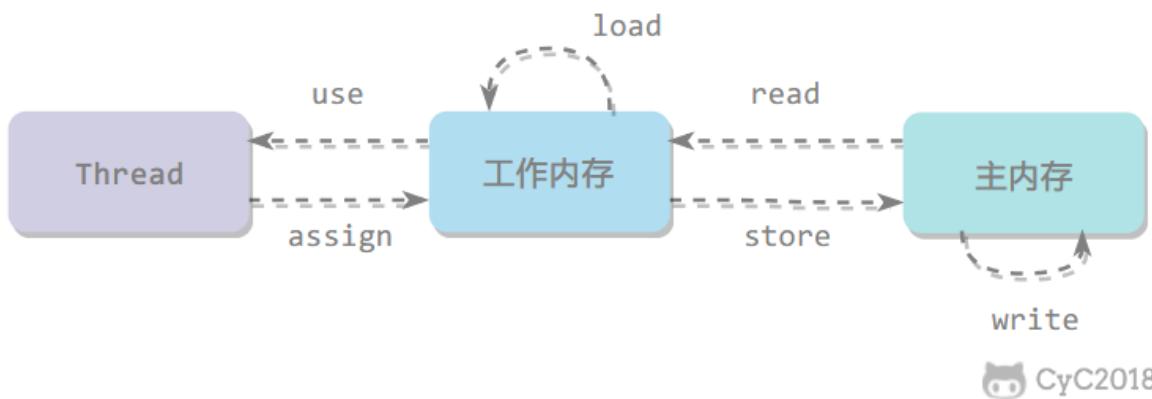
线程只能直接操作工作内存中的变量，不同线程之间的变量值传递需要通过主内存来完成。



CyC2018

## 内存间交互操作

Java 内存模型定义了 8 个操作来完成主内存和工作内存的交互操作。



CyC2018

- read: 把一个变量的值从主内存传输到工作内存中
- load: 在 read 之后执行, 把 read 得到的值放入工作内存的变量副本中
- use: 把工作内存中一个变量的值传递给执行引擎
- assign: 把一个从执行引擎接收到的值赋给工作内存的变量
- store: 把工作内存的一个变量的值传送到主内存中
- write: 在 store 之后执行, 把 store 得到的值放入主内存的变量中
- lock: 作用于主内存的变量
- unlock

## 为什么wait()一定要放在循环中

在多线程的编程实践中, wait()的使用方法如下:

```

synchronized (monitor) {
    // 判断条件谓词是否得到满足
    while(!locked) {
        // 等待唤醒
        monitor.wait();
    }
    // 处理其他的业务逻辑
}

```

那为什么非要while判断，而不采用if判断呢？如下：

```
synchronized (monitor) {  
    // 判断条件谓词是否得到满足  
    if(!locked) {  
        // 等待唤醒  
        monitor.wait();  
    }  
    // 处理其他的业务逻辑  
}
```

这是因为，如果采用if判断，当线程从wait中唤醒时，那么将直接执行处理其他业务逻辑的代码，但这时候可能出现另外一种可能，条件谓词已经不满足处理业务逻辑的条件了，从而出现错误的结果，于是有必要进行再一次判断，如下：

```
synchronized (monitor) {  
    // 判断条件谓词是否得到满足  
    if(!locked) {  
        // 等待唤醒  
        monitor.wait();  
        if(locked) {  
            // 处理其他的业务逻辑  
        } else {  
            // 跳转到monitor.wait();  
        }  
    }  
}
```

## 乐观锁的缺点

### 1 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。

### 2 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。

### 3 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，你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。

# ReentrantLock中lock()和tryLock()

---

- public void lock()

获取读取锁定。如果另一个线程没有保持写入锁定，则获取读取锁定并立即返回。如果另一个线程保持该写入锁定，出于线程调度目的，将禁用当前线程，并且在获取读取锁定之前，该线程将一直处于休眠状态。

- public boolean tryLock()

仅当写入锁定在调用期间未被另一个线程保持时获取读取锁定。

如果另一个线程没有保持写入锁定，则获取读取锁定并立即返回 true 值。即使已将此锁定设置为使用公平排序策略，但是调用 tryLock() 仍将立即获取读取锁定（如果有可用的），不管其他线程当前是否正在等待该读取锁定。在某些情况下，此“闯入”行为可能很有用，即使它会打破公平性也如此。如果希望遵守此锁定的公平设置，则使用 tryLock(0, TimeUnit.SECONDS)，它几乎是等效的（它也检测中断）。如果写入锁定被另一个线程保持，则此方法将立即返回 false 值。

**可见最大的区别，就是会不会被休眠等待。**

# Java基础

---

## 面向对象和面向过程的区别

---

### 面向过程

**优点：**性能比面向对象高。因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候（如：单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发）

**缺点：**没有面向对象易维护、易复用、易扩展

### 面向对象

**优点：**易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

**缺点：**性能比面向过程低

## Java面向对象编程三大特性: 封装 继承 多态

---

### 封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

### 继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

**关于继承如下 3 点请记住：**

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。

### 多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

好处：维护性、扩展性。

## Java程序初始化的顺序是怎么样的

---

在 Java 语言中，当实例化对象时，对象所在类的所有成员变量首先要进行初始化，只有当所有类成员完成初始化后，才会调用对象所在类的构造函数创建象。

### 初始化一般遵循3个原则：

- 静态对象（变量）优先于非静态对象（变量）初始化，静态对象（变量）只初始化一次，而非静态对象（变量）可能会初始化多次；
- 父类优先于子类进行初始化；
- 按照成员变量的定义顺序进行初始化。即使变量定义散布于方法定义之中，它们依然在任何方法（包括构造函数）被调用之前先初始化；

### 加载顺序

- 父类（静态变量、静态语句块）
- 子类（静态变量、静态语句块）
- 父类（实例变量、普通语句块）
- 父类（构造函数）
- 子类（实例变量、普通语句块）
- 子类（构造函数）

### 实例

```
class Base {  
    // 1.父类静态代码块  
    static {  
        System.out.println("Base static block!");  
    }  
    // 3.父类非静态代码块  
    {  
        System.out.println("Base block");  
    }  
    // 4.父类构造器  
    public Base() {  
        System.out.println("Base constructor!");  
    }  
}  
  
public class Derived extends Base {  
    // 2.子类静态代码块  
    static{  
        System.out.println("Derived static block!");  
    }  
    // 5.子类非静态代码块  
    {  
        System.out.println("Derived block!");  
    }  
    // 6.子类构造器  
    public Derived() {  
        System.out.println("Derived constructor!");  
    }  
    public static void main(String[] args) {  
        new Derived();  
    }  
}
```

结果是：

```
Base static block!
Derived static block!
Base block
Base constructor!
Derived block!
Derived constructor!
```

## String, StringBuffer and StringBuilder

### 1. 可变性

- String 不可变
- StringBuffer 和 StringBuilder 可变

### 2. 线程安全

- String 不可变，因此是线程安全的
- StringBuilder 不是线程安全的
- StringBuffer 是线程安全的，内部使用 synchronized 进行同步

## String Pool

字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程中将字符串添加到 String Pool 中。

当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等（使用 equals() 方法进行确定），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 方法取得一个字符串引用。intern() 首先把 s1 引用的字符串放到 String Pool 中，然后返回这个字符串引用。因此 s3 和 s4 引用的是同一个字符串。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);           // false
String s3 = s1.intern();
String s4 = s1.intern();
System.out.println(s3 == s4);           // true
```

如果是采用 "bbb" 这种字面量的形式创建字符串，会自动地将字符串放入 String Pool 中。

```
String s5 = "bbb";
String s6 = "bbb";
System.out.println(s5 == s6); // true
```

在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

- [StackOverflow : What is String interning?](#)
- [深入解析 String#intern](#)

## new String("abc")

使用这种方式一共会创建两个字符串对象（前提是 String Pool 中还没有 "abc" 字符串对象）。

- "abc" 属于字符串字面量，因此编译时期会在 String Pool 中创建一个字符串对象，指向这个 "abc" 字符串字面量；
- 而使用 new 的方式会在堆中创建一个字符串对象。

创建一个测试类，其 main 方法中使用这种方式来创建字符串对象。

```
public class NewStringTest {
    public static void main(String[] args) {
        String s = new String("abc");
    }
}
```

使用 javap -verbose 进行反编译，得到以下内容：

```
// ...
Constant pool:
// ...
#2 = class           #18          // java/lang/String
#3 = String          #19          // abc
// ...
#18 = Utf8           java/lang/String
#19 = Utf8           abc
// ...

public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
  0: new             #2          // class java/lang/String
  3: dup
  4: ldc             #3          // String abc
  6: invokespecial #4          // Method java/lang/String."<"
<init>":([Ljava/lang/String;)V
  9: astore_1
// ...
```

在 Constant Pool 中，#19 存储这字符串字面量 "abc"，#3 是 String Pool 的字符串对象，它指向 #19 这个字符串字面量。在 main 方法中，0: 行使用 new #2 在堆中创建一个字符串对象，并且使用 ldc #3 将 String Pool 中的字符串对象作为 String 构造函数的参数。

以下是 String 构造函数的源码，可以看到，在将一个字符串对象作为另一个字符串对象的构造函数参数时，并不会完全复制 value 数组内容，而是都会指向同一个 value 数组。

```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

## 接口和抽象类的区别是什么？

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现)，而抽象类可以有非抽象的方法。
2. 接口中除了 static、final 变量，不能有其他变量，而抽象类中则不一定。

3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过extends关键字扩展多个接口。
4. 接口方法默认修饰符是public，抽象方法可以有public、protected和default这些修饰符（抽象方法就是为了被重写所以不能使用private关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：在JDK8中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。

## == 与 equals(重要)

---

**==** : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型 == 比较的是值，引用数据类型 == 比较的是内存地址)。

**equals()** : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
- 情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

**说明：**

- String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。
- 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

## hashCode 与 equals (重要)

---

面试官可能会问你：“你重写过 hashCode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

### hashCode()介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

### 为什么要有 hashCode

我们先以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashCode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashCode 值作比较，如果没有相符的 hashCode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashCode 值的对象，这时会调用 equals() 方法来检查 hashCode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head first java》第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

通过我们可以看出：`hashCode()` 的作用就是**获取哈希码**，也称为散列码；它实际上是返回一个int整数。这个**哈希码的作用**是确定该对象在哈希表中的索引位置。`hashCode()` 在散列表中才有用，在其它情况下没用。在散列表中`hashCode()` 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。

## hashCode()与equals()的相关规定

1. 如果两个对象相等，则`hashCode`一定也是相同的
2. 两个对象相等，对两个对象分别调用`equals`方法都返回true
3. 两个对象有相同的`hashCode`值，它们也不一定是相等的
4. 因此，`equals`方法被覆盖过，则`hashCode`方法也必须被覆盖
5. `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写`hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

## 为什么 Java 中只有值传递？

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。**按值调用(call by value)**表示方法接收的是调用者提供的值，而**按引用调用 (call by reference)**表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。它用来描述各种程序设计语言（不只是Java）中方法参数传递方式。

Java程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

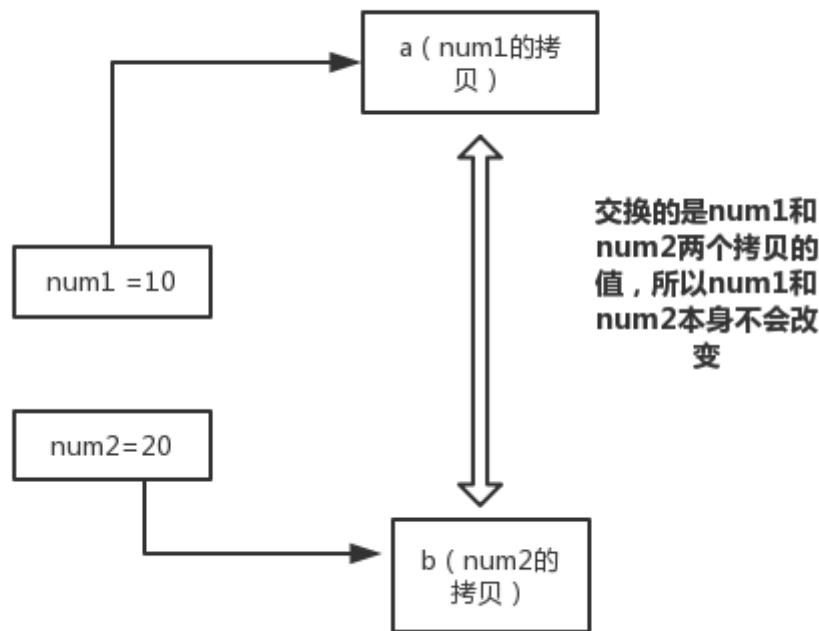
### example 1

```
public static void main(String[] args) {  
    int num1 = 10;  
    int num2 = 20;  
    swap(num1, num2);  
    System.out.println("num1 = " + num1);  
    System.out.println("num2 = " + num2);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
    System.out.println("a = " + a);  
    System.out.println("b = " + b);  
}
```

结果：

```
a = 20  
b = 10  
num1 = 10  
num2 = 20
```

解析：



在swap方法中，a、b的值进行交换，并不会影响到 num1、num2。因为，a、b中的值，只是从 num1、num2 的复制过来的。也就是说，a、b相当于num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样，请看 example2.

## example 2

```

public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5 };
    System.out.println(arr[0]);
    change(arr);
    System.out.println(arr[0]);
}

public static void change(int[] array) {
    // 将数组的第一个元素变为0
    array[0] = 0;
}

```

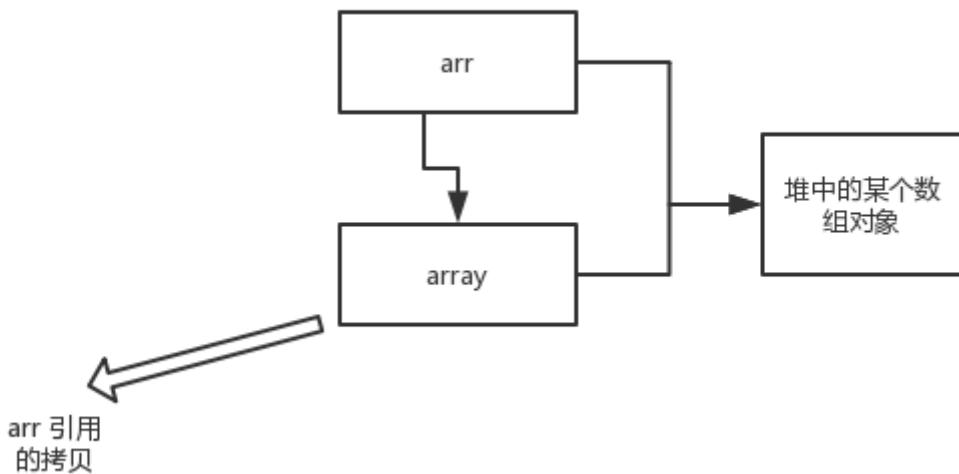
结果：

```

1
0

```

解析：



array 被初始化 arr 的拷贝也是一个对象的引用，也就是说 array 和 arr 指向的是同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal）提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为Java程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

### example 3

```
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s1 = new Student("小张");
        Student s2 = new Student("小李");
        Test.swap(s1, s2);
        System.out.println("s1:" + s1.getName());
        System.out.println("s2:" + s2.getName());
    }

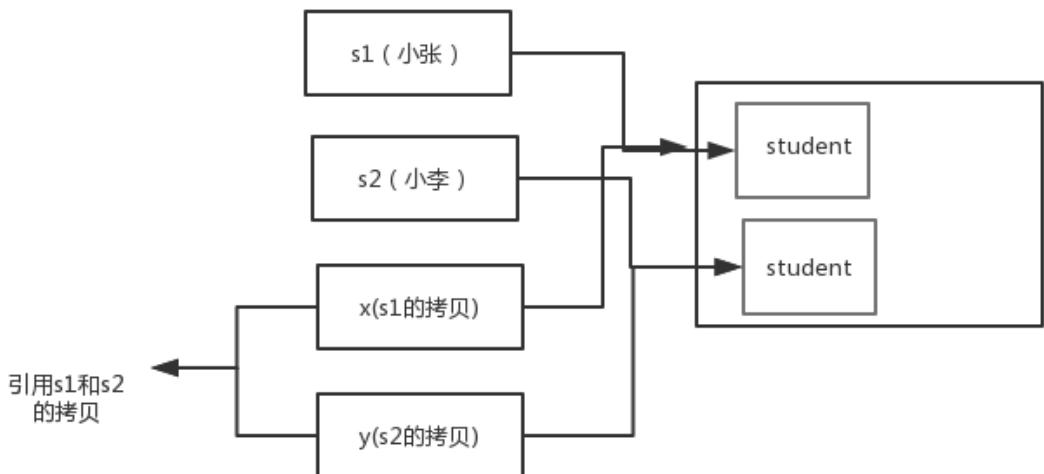
    public static void swap(Student x, Student y) {
        Student temp = x;
        x = y;
        y = temp;
        System.out.println("x:" + x.getName());
        System.out.println("y:" + y.getName());
    }
}
```

结果：

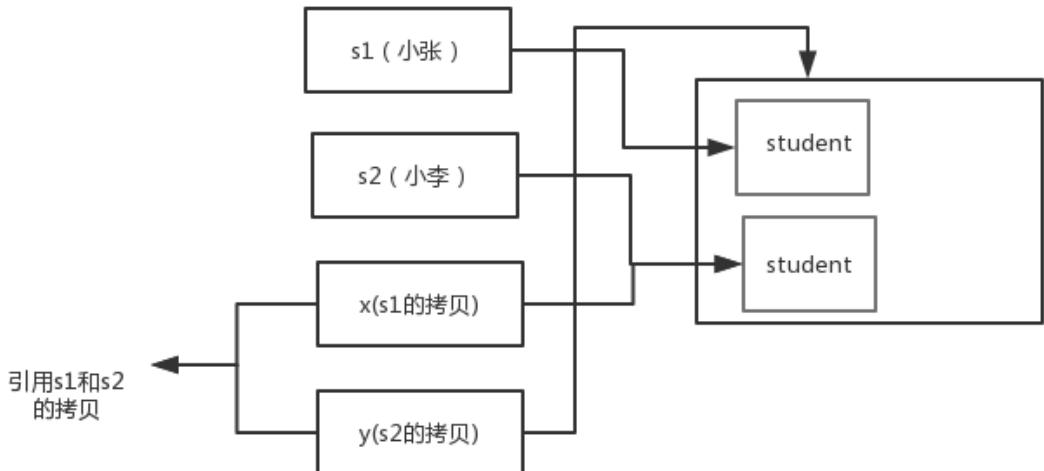
```
x:小李
y:小张
s1:小张
s2:小李
```

解析：

交换之前：



交换之后：



通过上面两张图可以很清晰的看出：方法并没有改变存储在变量 s1 和 s2 中的对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝

## 总结

Java程序设计语言对对象采用的不是引用调用，实际上，对象引用是按值传递的。

下面再总结一下Java中方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能让对象参数引用一个新的对象。

## 反射

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为Java语言的反射机制。

当我们的程序在运行时，需要动态的加载一些类，这些类可能之前用不到所以不用加载到 JVM，而是在运行时根据需要才加载。

每个类都有一个 **Class** 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件内容保存着 Class 对象。

类加载相当于 Class 对象的加载，类在第一次使用时才动态加载到 JVM 中。也可以使用 `Class.forName("com.mysql.jdbc.Driver")` 这种方式来控制类的加载，该方法会返回一个 Class 对象。

反射可以提供运行时的类信息，并且这个类可以在运行时才加载进来，甚至在编译时期该类的 .class 不存在也可以加载进来。

Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类：

- **Field**：可以使用 `get()` 和 `set()` 方法读取和修改 Field 对象关联的字段；
- **Method**：可以使用 `invoke()` 方法调用与 Method 对象关联的方法；
- **Constructor**：可以用 Constructor 创建新的对象。

### 反射的优点：

- **可扩展性**：应用程序可以利用全限定名创建可扩展对象的实例，来使用来自外部的用户自定义类。
- **类浏览器和可视化开发环境**：一个类浏览器需要可以枚举类的成员。可视化开发环境（如 IDE）可以从利用反射中可用的类型信息中受益，以帮助程序员编写正确的代码。
- **调试器和测试工具**：调试器需要能够检查一个类里的私有成员。测试工具可以利用反射来自动地调用类里定义的可被发现的 API 定义，以确保一组测试中有较高的代码覆盖率。

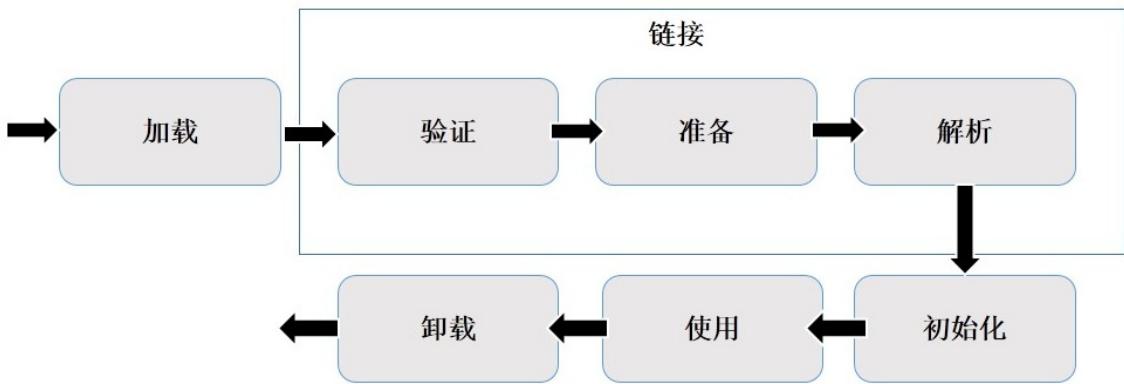
### 反射的缺点：

尽管反射非常强大，但也不能滥用。如果一个功能可以不用反射完成，那么最好就不用。在我们使用反射技术时，下面几条内容应该牢记于心。

- **性能开销**：反射涉及了动态类型的解析，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。
- **安全限制**：使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如 Applet，那么这就是个问题了。
- **内部暴露**：由于反射允许代码执行一些在正常情况下不被允许的操作（比如访问私有的属性和方法），所以使用反射可能会导致意料之外的副作用，这可能导致代码功能失调并破坏可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

## 反射中**Class.forName()**和**ClassLoader.loadClass()**的区别

### Java类装载过程



**装载：**通过类的全限定名获取二进制字节流，将二进制字节流转换成方法区中的运行时数据结构，在内存中生成java.lang.Class对象；

**链接：**执行下面的校验、准备和解析步骤，其中解析步骤是可以选择的；

**校验：**检查导入类或接口的二进制数据的正确性；（文件格式验证，元数据验证，字节码验证，符号引用验证）

**准备：**给类的静态变量分配并初始化存储空间；

**解析：**将常量池中的符号引用转成直接引用；

**初始化：**激活类的静态变量的初始化Java代码和静态Java代码块，并初始化程序员设置的变量值。

### 分析 Class.forName()和ClassLoader.loadClass()

Class.forName(className)方法，内部实际调用的方法是

Class.forName(className,true,classloader);

第2个boolean参数表示类是否需要初始化，Class.forName(className)默认是需要初始化。

一旦初始化，就会触发目标对象的 static块代码执行，static参数也也会被再次初始化。

ClassLoader.loadClass(className)方法，内部实际调用的方法是

ClassLoader.loadClass(className,false);

第2个 boolean参数，表示目标对象是否进行链接，false表示不进行链接，由上面介绍可以知道不进行链接意味着不进行包括初始化等一系列步骤，那么静态块和静态对象就不会得到执行

### 数据库链接为什么使用Class.forName(className)

JDBC Driver源码如下，因此使用Class.forName(className)才能在反射回去类的时候执行static块。

```

static {
    try {
        java.sql.DriverManager.registerDriver(new Driver());
    } catch (SQLException e) {
        throw new RuntimeException("Can't register driver!");
    }
}

```

## 序列化和反序列化

### 继承Serializable接口

`objectOutputStream` 的 `writeObject()` 方法。其部分源码如下：

```
if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {
    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
```

也就是说，`ObjectOutputStream` 在序列化的时候，会判断被序列化的对象是哪一种类型，字符串？数组？枚举？还是 `Serializable`，如果全都不是的话，抛出 `NotSerializableException`。

### 序列化ID的作用

序列化ID起着关键的作用，java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。反序列化时，JVM会把传来的字节流中的serialVersionUID与本地实体类中的serialVersionUID进行比较，如果相同则认为是一致的，便可以进行反序列化，否则就会报序列化版本不一致的异常。

## 泛型

### 10道Java泛型面试题

- Java中的泛型是什么？使用泛型的好处是什么？

这是在各种Java泛型面试中，一开场你就会被问到的问题中的一个，主要集中在初级和中级面试中。那些拥有Java1.4或更早版本的开发背景的人都知道，在集合中存储对象并在使用前进行类型转换是多么的不方便。泛型防止了那种情况的发生。它提供了编译期的类型安全，确保你只能把正确类型的对象放入集合中，避免了在运行时出现ClassCastException。

- Java的泛型是如何工作的？什么是类型擦除？

这是一道更好的泛型面试题。泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。根据你对这个泛型问题的回答情况，你会得到一些后续提问，比如为什么泛型是由类型擦除来实现的或者给你展示一些会导致编译器出错的错误泛型代码。请阅读我的Java中泛型是如何工作的来了解更多信息。

- 什么是泛型中的限定通配符和非限定通配符？

这是另一个非常流行的Java泛型面试题。限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表示了非限定通配符，因为<?>可以用任意类型来替代。更多信息请参阅我的文章泛型中限定通配符和非限定通配符之间的区别。

- List<? extends T>和List <? super T>之间有什么区别？

这和上一个面试题有联系，有时面试官会用这个问题来评估你对泛型的理解，而不是直接问你什么是限定通配符和非限定通配符。这两个List的声明都是限定通配符的例子，List<? extends T>可以接受任何继承自T的类型的List，而List<? super T>可以接受任何T的父类构成的List。例如List<? extends Number>可以接受List或List。在本段出现的连接中可以找到更多信息。

- 如何编写一个泛型方法，让它能接受泛型参数并返回泛型类型？

编写泛型方法并不困难，你需要用泛型类型来替代原始类型，比如使用T, E or K,V等被广泛认可的类型占位符。泛型方法的例子请参阅Java集合类框架。最简单的情况下，一个泛型方法可能会像这样：

```
public V put(K key, V value) {  
    return cache.put(key, value);  
}
```

- Java中如何使用泛型编写带有参数的类？

这是上一道面试题的延伸。面试官可能会要求你用泛型编写一个类型安全的类，而不是编写一个泛型方法。关键仍然是使用泛型类型来代替原始类型，而且要使用JDK中采用的标准占位符。

- 编写一段泛型程序来实现LRU缓存？

对于喜欢Java编程的人来说这相当于是一次练习。给你个提示，LinkedHashMap可以用来实现固定大小的LRU缓存，当LRU缓存已经满了的时候，它会把最老的键值对移出缓存。LinkedHashMap提供了一个称为removeEldestEntry()的方法，该方法会被put()和putAll()调用来删除最老的键值对。当然，如果你已经编写了一个可运行的JUnit测试，你也可以随意编写你自己的实现代码。

- 你可以把List传递给一个接受List参数的方法吗？

对任何一个不太熟悉泛型的人来说，这个Java泛型题目看起来令人疑惑，因为乍看起来String是一种Object，所以List应当可以用在需要List的地方，但是事实并非如此。真这样做的话会导致编译错误。如果你再深一步考虑，你会发现Java这样做是有意义的，因为List可以存储任何类型的对象包括String, Integer等等，而List却只能用来存储Strings。

```
List<Object> objectList;  
List<String> stringList;  
objectList = stringList; //compilation error incompatible types
```

- Array中可以用泛型吗？

这可能是Java泛型面试题中最简单的一个了，当然前提是你要知道Array事实上并不支持泛型，这也是为什么Joshua Bloch在Effective Java一书中建议使用List来代替Array，因为List可以提供编译期的类型安全保证，而Array却不能。

- 如何阻止Java中的类型未检查的警告？

如果你把泛型和原始类型混合起来使用，例如下列代码，Java 5的javac编译器会产生类型未检查的警告，例如

```
List<String> rawList = new ArrayList()
```

## 异常

- 如果你不写捕获异常的具体代码的话，你可以使用 throws Exception 的形式，把异常再次向上抛出，交给JVM(Java虚拟机)可以捕获，这是一种比较省事的办法
- 如果你想亲编写处理异常的代码的话，可以使用try{ }catch(){ } 的形式，进行捕获，一旦程序发生异常，它就会按照你catch{ }块编写的代码去执行

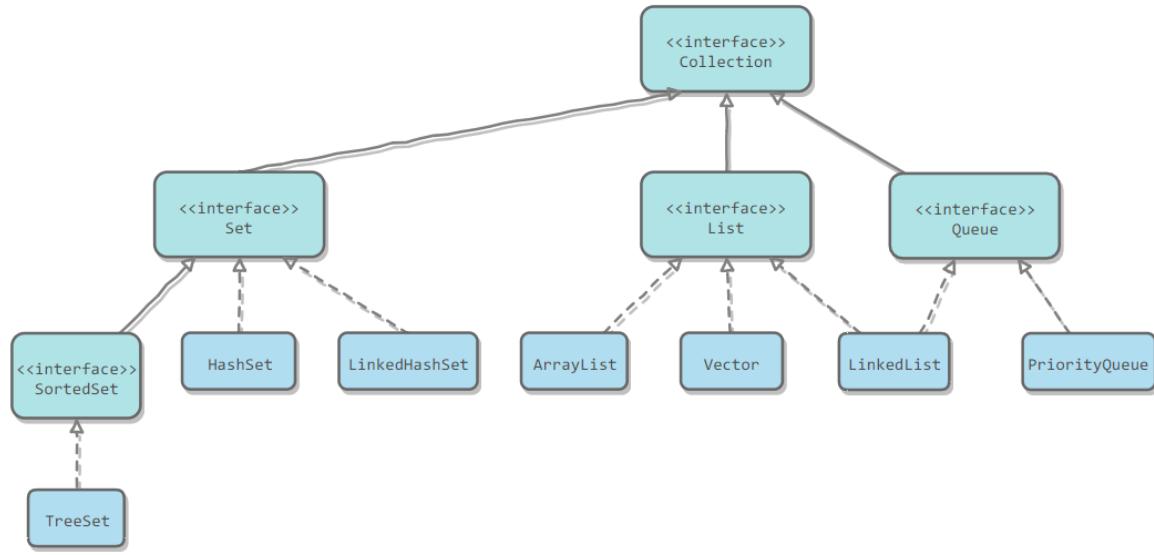
ClassNotFoundException	NoClassDefFoundError
从java.lang.Exception继承，是一个Exception类型	从java.lang.Error继承，是一个Error类型
当动态加载Class的时候找不到类会抛出该异常	当编译成功以后执行过程中 Class找不到导致抛出该错误
一般在执行Class.forName()、 ClassLoader.loadClass()或 ClassLoader.findSystemClass()的时候抛出	由JVM的运行时系统抛出

# Java容器

## 概览

容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表。

## Collection



CyC2018

## Set

- `TreeSet`: 基于红黑树实现，支持有序性操作，例如根据一个范围查找元素的操作。但是查找效率不如 `HashSet`，`HashSet` 查找的时间复杂度为  $O(1)$ ，`TreeSet` 则为  $O(\log N)$ 。
- `HashSet`: 基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 `Iterator` 遍历 `HashSet` 得到的结果是不确定的。
- `LinkedHashSet`: 具有 `HashSet` 的查找效率，且内部使用双向链表维护元素的插入顺序。

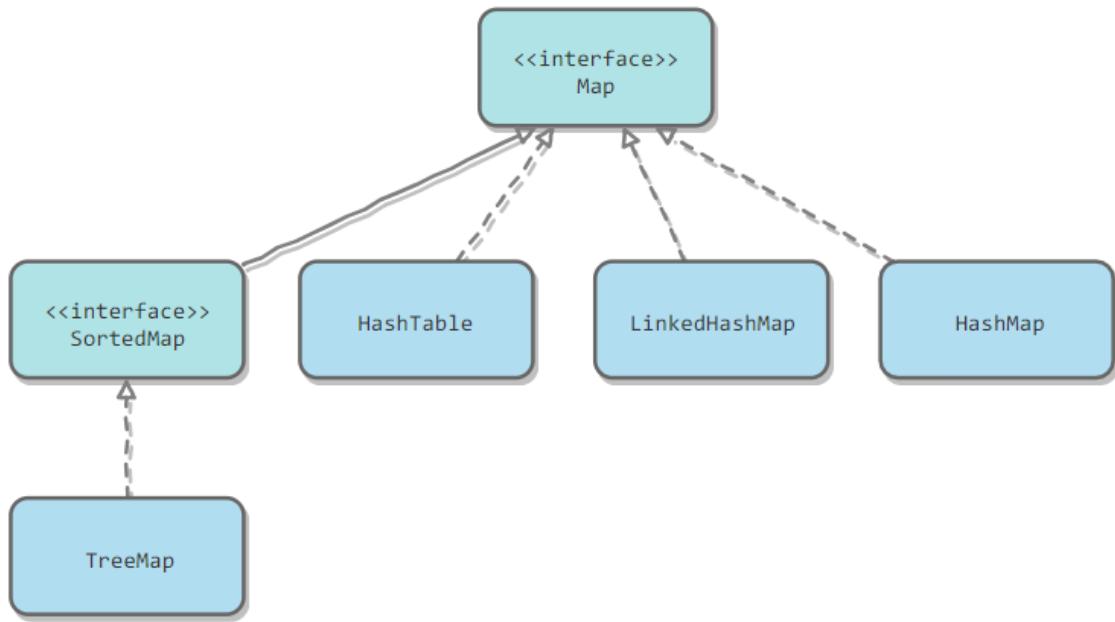
## List

- `ArrayList`: 基于动态数组实现，支持随机访问。
- `Vector`: 和 `ArrayList` 类似，但它是线程安全的。
- `LinkedList`: 基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，`LinkedList` 还可以用作栈、队列和双向队列。

## Queue

- `LinkedList`: 可以用它来实现双向队列。
- `PriorityQueue`: 基于堆结构实现，可以用它来实现优先队列。

## Map



CyC2018

- TreeMap: 基于红黑树实现。
- HashMap: 基于哈希表实现。
- HashTable: 和 HashMap 类似, 但它是线程安全的, 这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类, 不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全, 并且 ConcurrentHashMap 的效率会更高, 因为 ConcurrentHashMap 引入了分段锁。
- LinkedHashMap: 使用双向链表来维护元素的顺序, 顺序为插入顺序或者最近最少使用 (LRU) 顺序。

## 比较器Comparable和Comparator的区别

- Comparable接口实际上是出自java.lang包 它有一个 `compareTo(Object obj)` 方法用来排序
- Comparator接口实际上是出自 java.util包它有一个 `compare(Object obj1, Object obj2)` 方法用来排序

一般我们需要对一个集合使用自定义排序时, 我们就要重写 `compareTo()` 方法或 `compare()` 方法, 当我们需要对某一个集合实现两种排序方式, 比如一个song对象中的歌名和歌手名分别采用一种排序方法的话, 我们可以重写 `compareTo()` 方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序, 第二种代表我们只能使用两个参数版的 `collections.sort()` .

## Comparator定制排序

```

ArrayList<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(-1);
arrayList.add(3);
arrayList.add(3);
arrayList.add(-5);
arrayList.add(7);
arrayList.add(4);
arrayList.add(-9);
arrayList.add(-7);
System.out.println("原始数组:");
System.out.println(arrayList);
// void reverse(List list): 反转
  
```

```

        collections.reverse(arrayList);
        System.out.println("collections.reverse(arrayList):");
        System.out.println(arrayList);

        // void sort(List list),按自然排序的升序排序
        Collections.sort(arrayList);
        System.out.println("Collections.sort(arrayList):");
        System.out.println(arrayList);
        // 定制排序的用法
        Collections.sort(arrayList, new Comparator<Integer>() {

            @Override
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
        });
        System.out.println("定制排序后: ");
        System.out.println(arrayList);
    }
}

```

Output:

```

原始数组:
[-1, 3, 3, -5, 7, 4, -9, -7]
Collections.reverse(arrayList):
[-7, -9, 4, 7, -5, 3, 3, -1]
Collections.sort(arrayList):
[-9, -7, -5, -1, 3, 3, 4, 7]
定制排序后:
[7, 4, 3, 3, -1, -5, -7, -9]

```

## 重写compareTo方法实现按年龄来排序

```

// person对象没有实现Comparable接口, 所以必须实现, 这样才不会出错, 才可以使treemap中的数据
按顺序排列
// 前面一个例子的String类已经默认实现了Comparable接口, 详细可以查看String类的API文档, 另外
其他
// 像Integer类等都已经实现了Comparable接口, 所以不需要另外实现了

public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

```

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

/**
 * TODO重写compareTo方法实现按年龄来排序
 */
@Override
public int compareTo(Person o) {
    // TODO Auto-generated method stub
    if (this.age > o.getAge()) {
        return 1;
    } else if (this.age < o.getAge()) {
        return -1;
    }
    return age;
}
}

public static void main(String[] args) {
    TreeMap<Person, String> pdata = new TreeMap<Person, String>();
    pdata.put(new Person("张三", 30), "zhangsan");
    pdata.put(new Person("李四", 20), "lisi");
    pdata.put(new Person("王五", 10), "wangwu");
    pdata.put(new Person("小红", 5), "xiaohong");
    // 得到key的值的同时得到key所对应的值
    Set<Person> keys = pdata.keySet();
    for (Person key : keys) {
        System.out.println(key.getAge() + "—" + key.getName());
    }
}
}

```

Output:

```

5-小红
10-王五
20-李四
30-张三

```

## 快速失败和安全失败

我们都接触 HashMap、ArrayList 这些集合类，这些在 java.util 包的集合类就都是快速失败的；而 java.util.concurrent 包下的类都是安全失败，比如： ConcurrentHashMap。

### 快速失败 (fail-fast)

在使用迭代器对集合对象进行遍历的时候，如果 A 线程正在对集合进行遍历，此时 B 线程对集合进行修改（增加、删除、修改），或者 A 线程在遍历过程中对集合进行修改，都会导致 A 线程抛出 ConcurrentModificationException 异常。

具体效果我们看下代码：

```

HashMap hashMap = new HashMap();
hashMap.put("不只Java-1", 1);
hashMap.put("不只Java-2", 2);
hashMap.put("不只Java-3", 3);
Set set = hashMap.entrySet();
Iterator iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
    hashMap.put("下次循环会抛异常", 4);
    System.out.println("此时 hashMap 长度为" + hashMap.size());
}

```

执行后的效果如下图：

```

E:\JDK\bin\java "-javaagent:D:\Program Files\JetBrains\IntelliJ IDEA
不只Java-1=1
此时 hashMap 长度为4
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1429)
    at java.util.HashMap$EntryIterator.next(HashMap.java:1463)
    at java.util.HashMap$EntryIterator.next(HashMap.java:1461)
    at com.czd.hash.main(hash.java:38)

```

为什么在用迭代器遍历时，修改集合就会抛异常时？

原因是迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。

每当迭代器使用 hashNext()/next() 遍历下一个元素之前，都会检测 modCount 变量是否为 expectedModCount 值，是的话就返回遍历；否则抛出异常，终止遍历。

### 安全失败 (fail-safe)

明白了什么是快速失败之后，安全失败也是非常好理解的。

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，故不会抛 ConcurrentModificationException 异常

我们上代码看下是不是这样

```

ConcurrentHashMap concurrentHashMap = new ConcurrentHashMap();
concurrentHashMap.put("不只Java-1", 1);
concurrentHashMap.put("不只Java-2", 2);
concurrentHashMap.put("不只Java-3", 3);
Set set = concurrentHashMap.entrySet();
Iterator iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
    concurrentHashMap.put("下次循环正常执行", 4);
}
System.out.println("程序结束");

```

运行效果如下，的确不会抛异常，程序正常执行。

Connected to the target VM, add  
Disconnected from the target VM  
不只Java-1=1  
不只Java-2=2  
不只Java-3=3  
程序结束

最后说明一下，**快速失败和安全失败是对迭代器而言的**。并发环境下建议使用java.util.concurrent包下的容器类，除非没有修改操作。

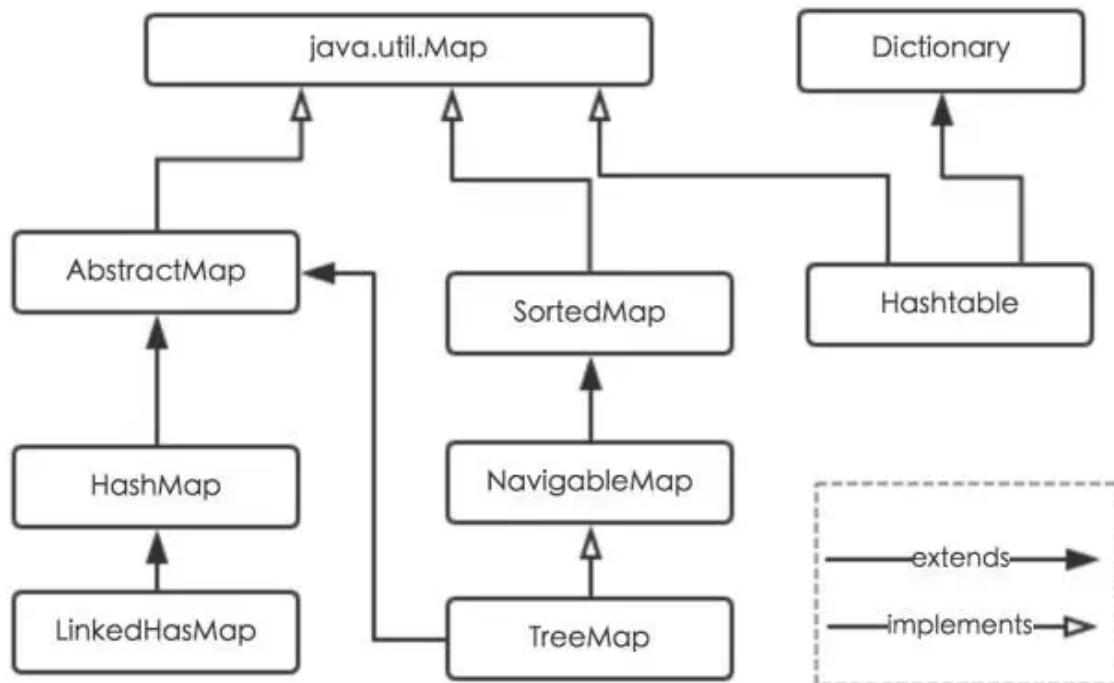
## HashMap

### 摘要

HashMap是Java程序员使用频率最高的用于映射(键值对)处理的数据类型。随着JDK (Java Developmet Kit) 版本的更新，JDK1.8对HashMap底层的实现进行了优化，例如引入红黑树的数据结构和扩容的优化等。本文结合JDK1.7和JDK1.8的区别，深入探讨HashMap的结构实现和功能原理。

### 简介

Java为数据结构中的映射定义了一个接口java.util.Map，此接口主要有四个常用的实现类，分别是HashMap、Hashtable、LinkedHashMap和TreeMap，类继承关系如下图所示：



下面针对各个实现类的特点做一些说明：

(1) `HashMap`: 它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。`HashMap`最多只允许一条记录的键为null，允许多条记录的值为null。`HashMap`非线程安全，即任一时刻可以有多个线程同时写`HashMap`，可能会导致数据的不一致。如果需要满足线程安全，可以用Collections的`synchronizedMap`方法使`HashMap`具有线程安全的能力，或者使用`ConcurrentHashMap`。

(2) `Hashtable`: `Hashtable`是遗留类，很多映射的常用功能与`HashMap`类似，不同的是它承自`Dictionary`类，并且是线程安全的，任一时间只有一个线程能写`Hashtable`，并发性不如`ConcurrentHashMap`，因为`ConcurrentHashMap`引入了分段锁。`Hashtable`不建议在新代码中使用，不需要线程安全的场合可以用`HashMap`替换，需要线程安全的场合可以用`ConcurrentHashMap`替换。

(3) LinkedHashMap: LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

(4) TreeMap: TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

对于上述四种Map类型的类，要求映射中的key是不可变对象。不可变对象是该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

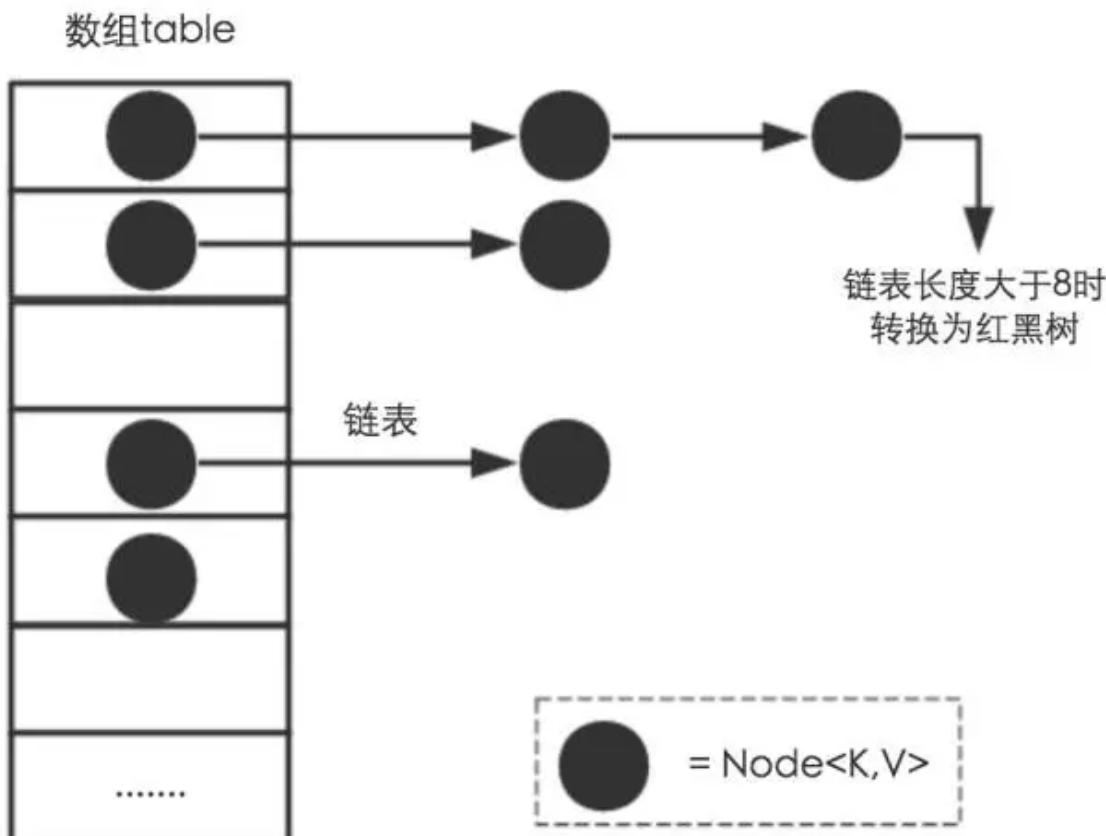
通过上面的比较，我们知道HashMap是Java的Map家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解HashMap的工作原理。

## 内部实现

搞清楚HashMap，首先需要知道HashMap是什么，即它的存储结构-字段；其次弄明白它能干什么，即它的功能实现-方法。下面我们针对这两个方面详细展开讲解。

### 存储结构-字段

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下所示。



这里需要讲明白两个问题：数据底层具体存储的是什么？这样的存储方式有什么优点呢？

(1) 从源码可知，HashMap类中有一个非常重要的字段，就是Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[JDK1.8]是何物。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;      //用来定位数组索引位置
    final K key;
    V value;
```

```

Node<K, V> next; //链表的下一个node

Node(int hash, K key, V value, Node<K, V> next) { ... }
public final K getKey() { ... }
public final V getValue() { ... }
public final String toString() { ... }
public final int hashCode() { ... }
public final V setValue(V newValue) { ... }
public final boolean equals(Object o) { ... }
}

```

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

(2) HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：

```
map.put("美团", "小美");
```

系统将调用"美团"这个key的hashCode()方法得到其hashCode值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组(Node[] table) 占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

```

int threshold; // 所能容纳的key-value对极限
final float loadFactor; // 负载因子
int modCount;
int size;

```

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不属于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考[http://blog.csdn.net/liuqiyao\\_01/article/details/14475159](http://blog.csdn.net/liuqiyao_01/article/details/14475159)，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。本文不再对红黑树展开讨论，想了解更多红黑树数据结构的工作原理可以参考[http://blog.csdn.net/v\\_july\\_v/article/details/6105630](http://blog.csdn.net/v_july_v/article/details/6105630)。

## 功能实现-方法

HashMap的内部功能实现很多，本文主要从根据key获取哈希桶数组索引位置、put方法的详细执行、扩容过程三个具有代表性的点深入展开讲解。

### 1、确定哈希桶数组索引位置

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，不用遍历链表，大大优化了查询的效率。HashMap定位数组索引位置，直接决定了hash方法的离散性能。先看看源码的实现(方法一+方法二)：

```
方法一：  
static final int hash(Object key) { //jdk1.8 & jdk1.7  
    int h;  
    // h = key.hashCode() 为第一步 取hashCode值  
    // h ^ (h >>> 16) 为第二步 高位参与运算  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}  
方法二：  
static int indexFor(int h, int length) { //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的  
    return h & (length-1); //第三步 取模运算  
}
```

这里的Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的那个索引处。

这个方法非常巧妙，它通过h & (table.length - 1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时，h & (length - 1)运算等价于对length取模，也就是h % length，但是&%具有更高的效率。

在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

下面举例说明下，n为table的长度。

`h=hashCode() : 1111 1111 1111 1111 1111 0000 1110 1010` 调用`hashCode()`

`h : 1111 1111 1111 1111 1111 0000 1110 1010`

`h >> 16 : 0000 0000 0000 0000 1111 1111 1111 1111`

计算Hash

`hash = h ^ (h >> 16) : 1111 1111 1111 1111 0000 1111 0001 0101`

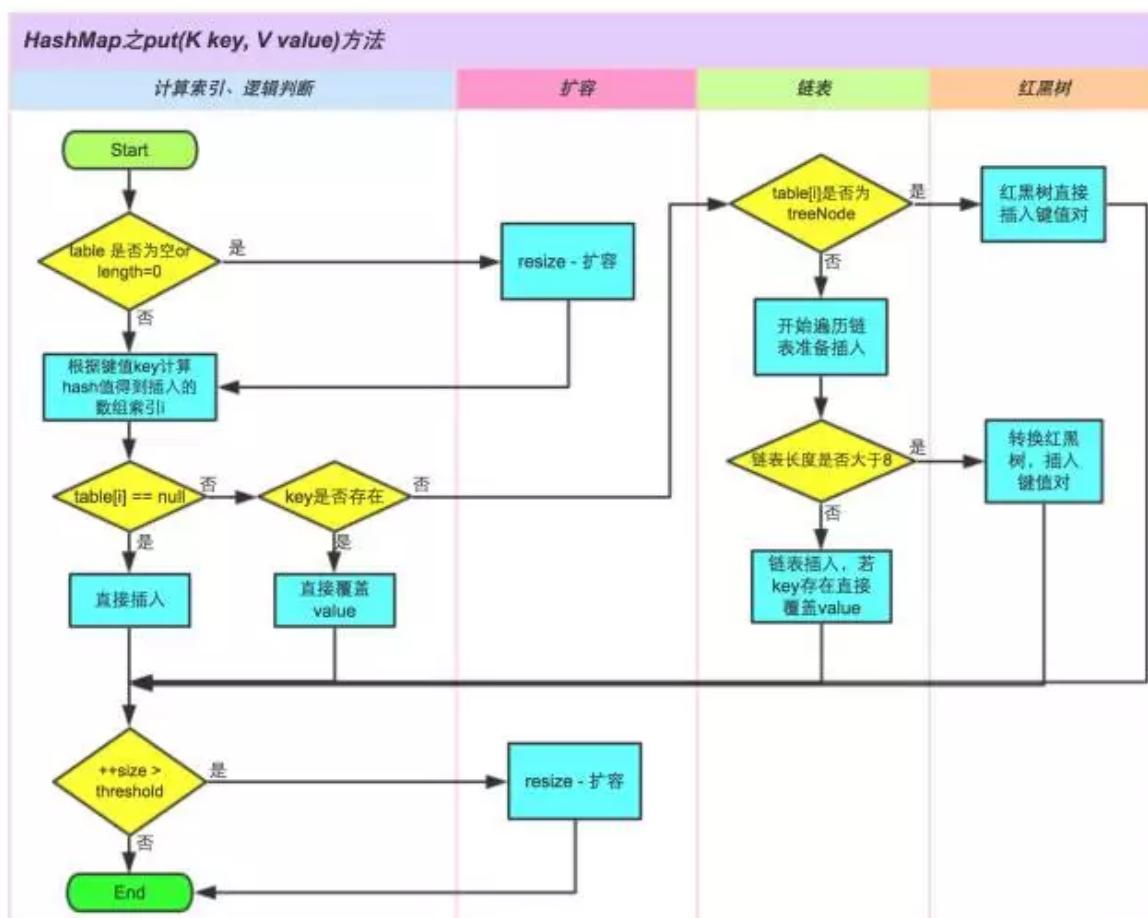
`(n - 1) & hash : 0000 0000 0000 0000 0000 0000 1111  
1111 1111 1111 1111 0000 1111 0001 0101`

计算下标

`0101 = 5`

## 2、分析HashMap的put方法

HashMap的put方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



①. 判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②. 根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③. 判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向

④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量

threshold，如果超过，进行扩容。

JDK1.8HashMap的put方法源码如下：

```
public V put(K key, V value) {
    // 对key的hashCode()做hash
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 步骤①: tab为空则创建
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 步骤②: 计算index，并对null做处理
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        // 步骤③: 节点key存在，直接覆盖value
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 步骤④: 判断该链为红黑树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 步骤⑤: 该链为链表
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    // 链表长度大于8转换为红黑树进行处理
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                // key已经存在直接覆盖value
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
            }
        }
    }
}
```

```

        return oldValue;
    }

    ++modCount;
    // 步骤⑥：超过最大容量 就扩容
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}

```

### 3、扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

我们分析下resize的源码，鉴于JDK1.8融入了红黑树，较复杂，为了便于理解我们仍然使用JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

```

void resize(int newCapacity) { //传入新的容量
    Entry[] oldTable = table; //引用扩容前的Entry数组
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)
        threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了
        return;
    }

    Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组
    transfer(newTable); //将数据转移到新的Entry数组里
    table = newTable; //HashMap的table属性引用新的Entry数组
    threshold = (int)(newCapacity * loadFactor); //修改阈值
}

```

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

```

void transfer(Entry[] newTable) {
    Entry[] src = table; //src引用了旧的Entry数组
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
        Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素
        if (e != null) {
            src[j] = null; //释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity); //重新计算每个元素在数组中的位置
                e.next = newTable[i]; //标记[1]
                newTable[i] = e; //将元素放在数组上
            }
        }
    }
}

```

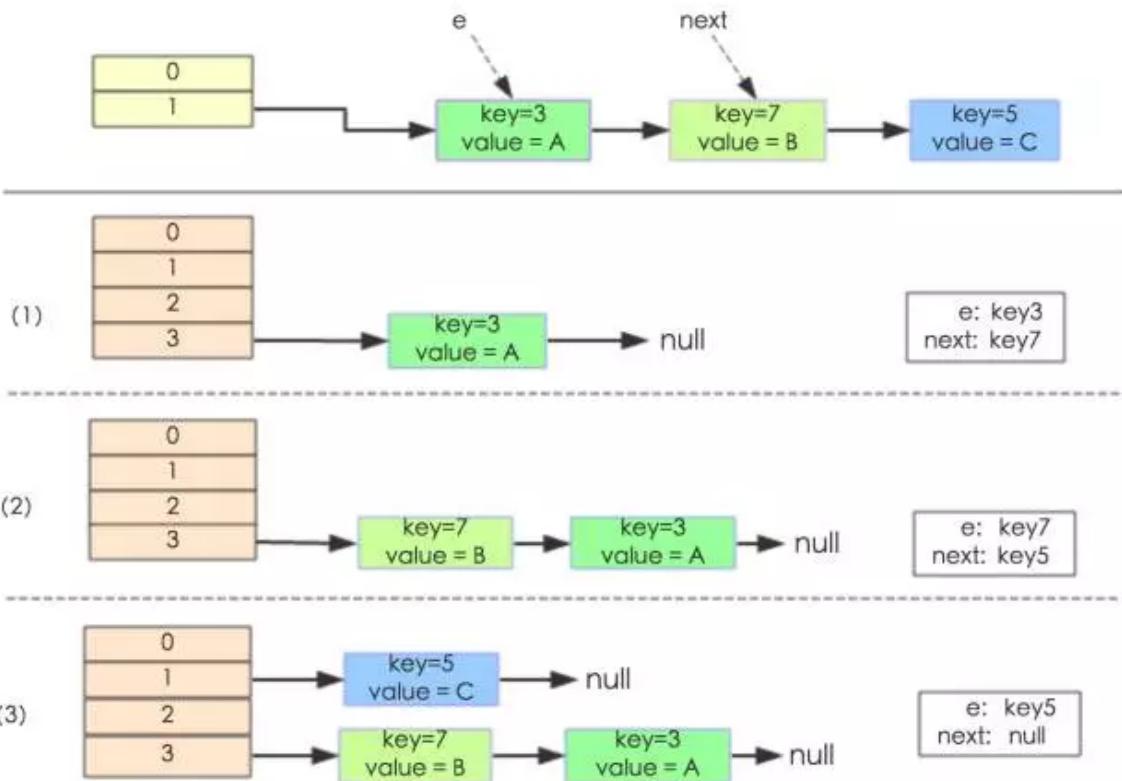
```

        e = next;           //访问下一个Entry链上的元素
    } while (e != null);
}
}
}

```

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话)，这一点和dk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

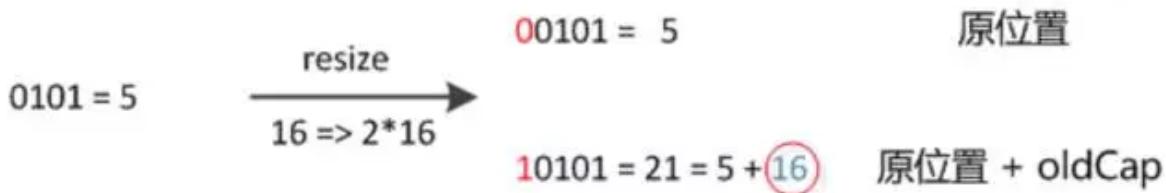
下面举个例子说明下扩容过程。假设了我们的hash算法就是简单的用key mod一下表的大小（也就是数组的长度）。其中的哈希桶数组table的size=2，所以key = 3、7、5，put顺序依次为 5、7、3。在mod 2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size大于table的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize成4，然后所有的Node重新rehash的过程。



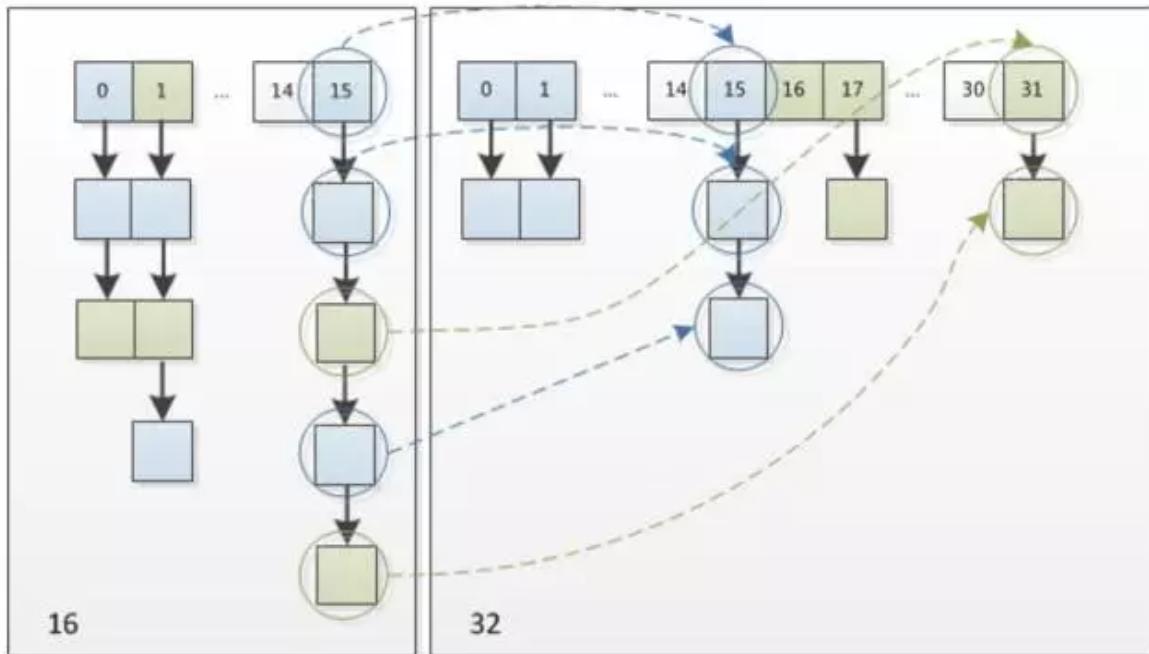
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图(a) 表示扩容前的key1和key2两种key确定索引位置的示例，图(b) 表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞。

### 线程安全性

在多线程使用场景中，应该尽量避免使用线程不安全的HashMap，而使用线程安全的ConcurrentHashMap。那么为什么说HashMap是线程不安全的，下面举例子说明在并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境)：

```

public class HashMapInfiniteLoop {
    private static HashMap<Integer, String> map = new HashMap<Integer, String>(2,
0.75f);
    public static void main(String[] args) {
        map.put(5, "C");

        new Thread("Thread1") {
            public void run() {
                map.put(7, "B");
                System.out.println(map);
            }
        }.start();
        new Thread("Thread2") {
            public void run() {
                map.put(3, "A");
            }
        }.start();
    }
}
  
```

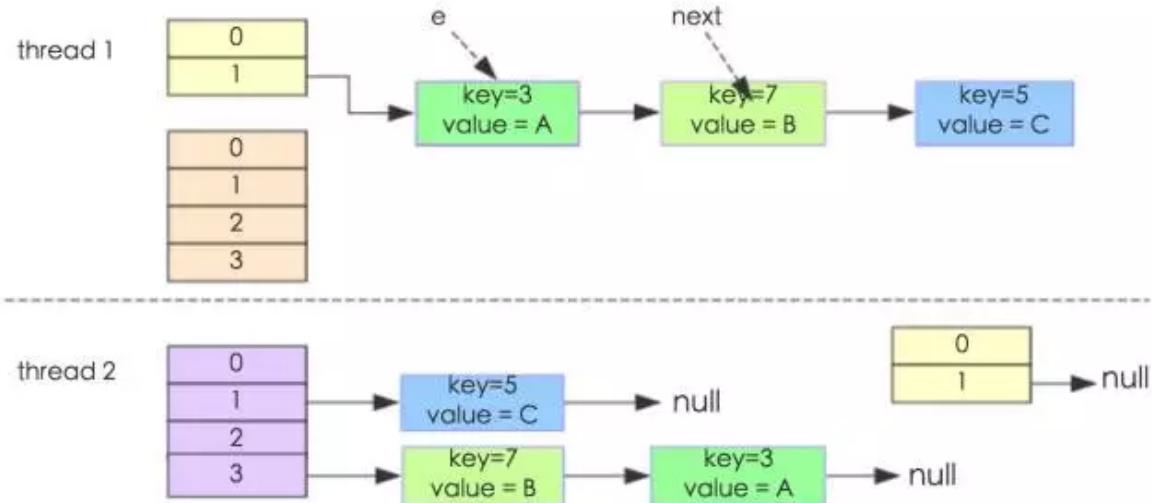
```

        System.out.println(map);
    };
    .start();
}
}

```

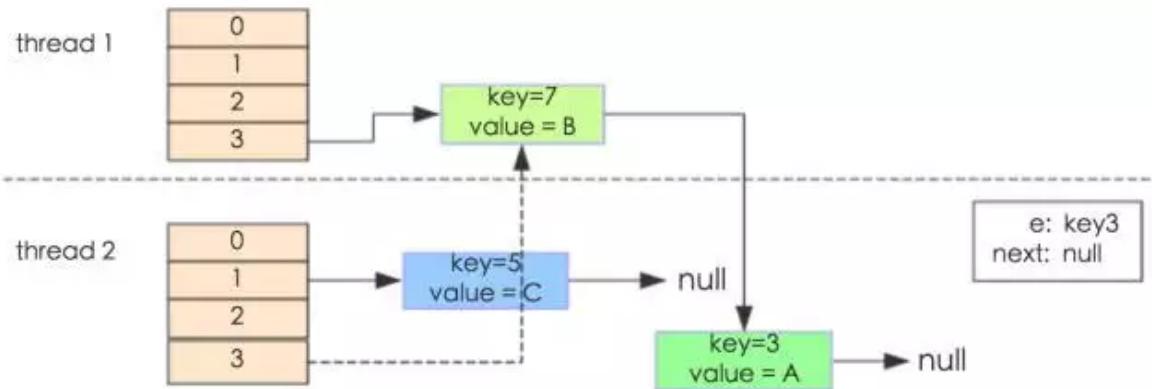
其中，map初始化为一个长度为2的数组，loadFactor=0.75，threshold=2\*0.75=1，也就是说当put第二个key的时候，map就需要进行resize。

通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;”这一行；然后放开线程2的断点，让线程2进行resize。结果如下图。

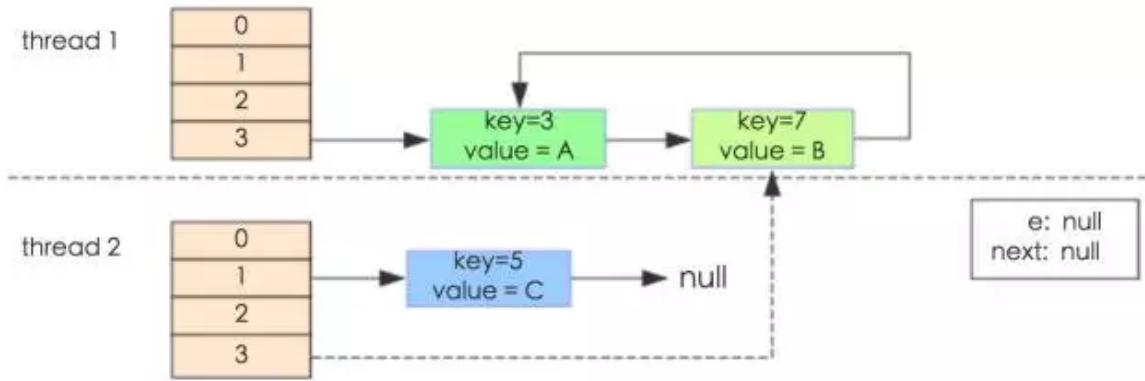


注意，Thread1的e指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

线程一被调度回来执行，先是执行 newTable[i] = e，然后是e = next，导致了e指向了key(7)，而下一次循环的next = e.next导致了next指向了key(3)。



e.next = newTable[i] 导致 key(3).next 指向了 key(7)。注意：此时的key(7).next 已经指向了key(3)，环形链表就这样出现了。



于是，当我们用线程一调用map.get(11)时，悲剧就出现了——Infinite Loop。

### JDK1.8与JDK1.7的性能对比

HashMap中，如果key经过hash算法得出的数组索引位置全部不相同，即Hash算法非常好，这样的话，getKey方法的时间复杂度就是O(1)，如果Hash算法技术的结果碰撞非常多，假如Hash算极其差，所有的Hash算法结果得出的索引位置一样，那样所有的键值对都集中到一个桶中，或者在一个链表中，或者在一个红黑树中，时间复杂度分别为O(n)和O(lgn)。鉴于JDK1.8做了多方面的优化，总体性能优于JDK1.7，下面我们从两个方面用例子证明这一点。

## 小结

- (1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。
- (2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。
- (3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。
- (4) JDK1.8引入红黑树大程度优化了HashMap的性能。
- (5) 还没升级JDK1.8的，现在开始升级吧。HashMap的性能提升仅仅是JDK1.8的冰山一角。

## 参考

- 1、JDK1.7&JDK1.8 源码。
- 2、CSDN博客频道，HashMap多线程死循环问题，2014。
- 3、红黑联盟，Java类集框架之HashMap(JDK1.8)源码剖析，2015。
- 4、CSDN博客频道，教你初步了解红黑树，2010。
- 5、Java Code Geeks，HashMap performance improvements in Java 8，2014。
- 6、Importnew，危险！在HashMap中将可变对象用作Key，2014。
- 7、CSDN博客频道，为什么一般hashtable的桶数会取一个素数，2013。

## LinkedHashMap

以下是使用 LinkedHashMap 实现的一个 LRU 缓存：

- 设定最大缓存空间 MAX\_ENTRIES 为 3；
- 使用 LinkedHashMap 的构造函数将 accessOrder 设置为 true，开启 LRU 顺序；
- 覆盖 removeEldestEntry() 方法实现，在节点多于 MAX\_ENTRIES 就会将最近最久未使用的数据移除。

```

class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private static final int MAX_ENTRIES = 3;

    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > MAX_ENTRIES;
    }

    LRUCache() {
        super(MAX_ENTRIES, 0.75f, true);
    }
}

public static void main(String[] args) {
    LRUCache<Integer, String> cache = new LRUCache<>();
    cache.put(1, "a");
    cache.put(2, "b");
    cache.put(3, "c");
    cache.get(1);
    cache.put(4, "d");
    System.out.println(cache.keySet());
}

```

[3, 1, 4]

## ConcurrentHashMap

### 存储结构

```

static final class HashEntry<K,V> {
    final int hash;
    final K key;
    volatile V value;
    volatile HashEntry<K,V> next;
}

```

ConcurrentHashMap 和 HashMap 实现上类似，最主要的差别是 ConcurrentHashMap 采用了分段锁 (Segment)，每个分段锁维护着几个桶 (HashEntry)，多个线程可以同时访问不同分段锁上的桶，从而使其并发度更高 (并发度就是 Segment 的个数)。

Segment 继承自 ReentrantLock。

```

static final class Segment<K,V> extends ReentrantLock implements Serializable {

    private static final long serialVersionUID = 2249069246763182397L;

    static final int MAX_SCAN_RETRIES =
        Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;

    transient volatile HashEntry<K,V>[] table;

    transient int count;

    transient int modCount;

    transient int threshold;
}

```

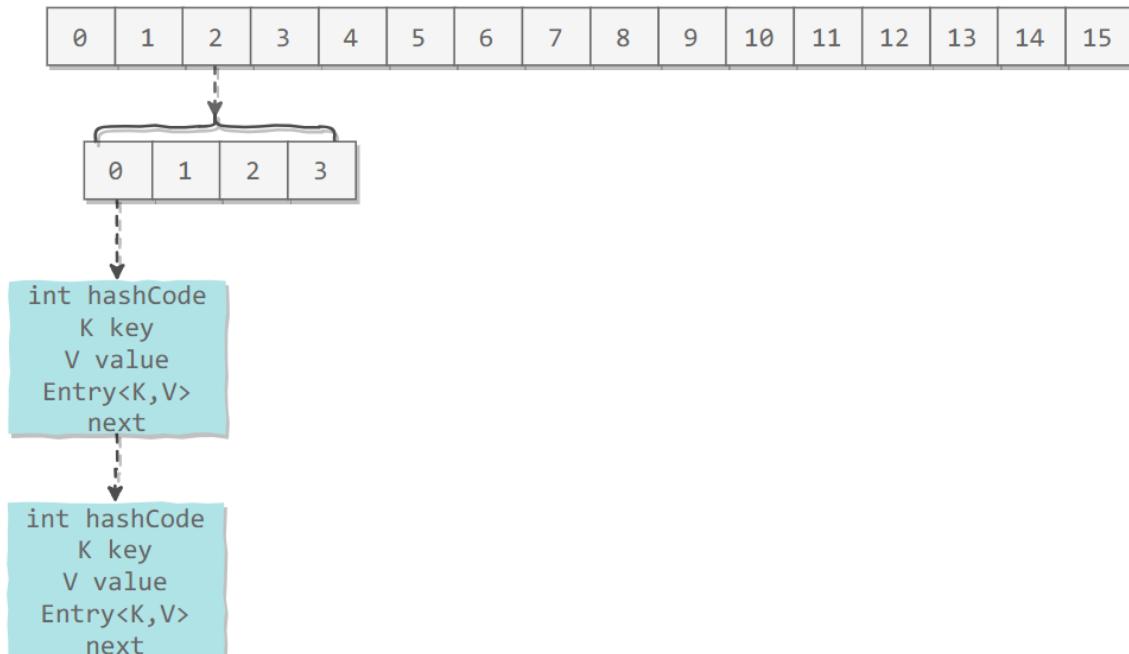
```

    final float loadFactor;
}
final Segment<K,V>[] segments;

```

默认的并发级别为 16，也就是说默认创建 16 个 Segment。

```
static final int DEFAULT_CONCURRENCY_LEVEL = 16;
```



CyC2018

## size 操作

每个 Segment 维护了一个 count 变量来统计该 Segment 中的键值对个数。

```

/**
 * The number of elements. Accessed only either within locks
 * or among other volatile reads that maintain visibility.
 */
transient int count;

```

在执行 size 操作时，需要遍历所有 Segment 然后把 count 累计起来。

ConcurrentHashMap 在执行 size 操作时先尝试不加锁，如果连续两次不加锁操作得到的结果一致，那么可以认为这个结果是正确的。

尝试次数使用 RETRIES\_BEFORE\_LOCK 定义，该值为 2，retries 初始值为 -1，因此尝试次数为 3。

如果尝试的次数超过 3 次，就需要对每个 Segment 加锁。

```

/**
 * Number of unsynchronized retries in size and containsvalue
 * methods before resorting to locking. This is used to avoid
 * unbounded retries if tables undergo continuous modification
 * which would make it impossible to obtain an accurate result.
 */
static final int RETRIES_BEFORE_LOCK = 2;

```

```

public int size() {
    // Try a few times to get accurate count. On failure due to
    // continuous async changes in table, resort to locking.
    final Segment<K,V>[] segments = this.segments;
    int size;
    boolean overflow; // true if size overflows 32 bits
    long sum;          // sum of modCounts
    long last = 0L;    // previous sum
    int retries = -1; // first iteration isn't retry
    try {
        for (;;) {
            // 超过尝试次数，则对每个 Segment 加锁
            if (retries++ == RETRIES_BEFORE_LOCK) {
                for (int j = 0; j < segments.length; ++j)
                    ensureSegment(j).lock(); // force creation
            }
            sum = 0L;
            size = 0;
            overflow = false;
            for (int j = 0; j < segments.length; ++j) {
                Segment<K,V> seg = segmentAt(segments, j);
                if (seg != null) {
                    sum += seg.modCount;
                    int c = seg.count;
                    if (c < 0 || (size += c) < 0)
                        overflow = true;
                }
            }
            // 连续两次得到的结果一致，则认为这个结果是正确的
            if (sum == last)
                break;
            last = sum;
        }
    } finally {
        if (retries > RETRIES_BEFORE_LOCK) {
            for (int j = 0; j < segments.length; ++j)
                segmentAt(segments, j).unlock();
        }
    }
    return overflow ? Integer.MAX_VALUE : size;
}

```

## JDK 1.8 的改动

JDK 1.7 使用分段锁机制来实现并发更新操作，核心类为 Segment，它继承自重入锁 ReentrantLock，并发度与 Segment 数量相等。

JDK 1.8 使用了 CAS 操作来支持更高的并发度，在 CAS 操作失败时使用内置锁 synchronized。

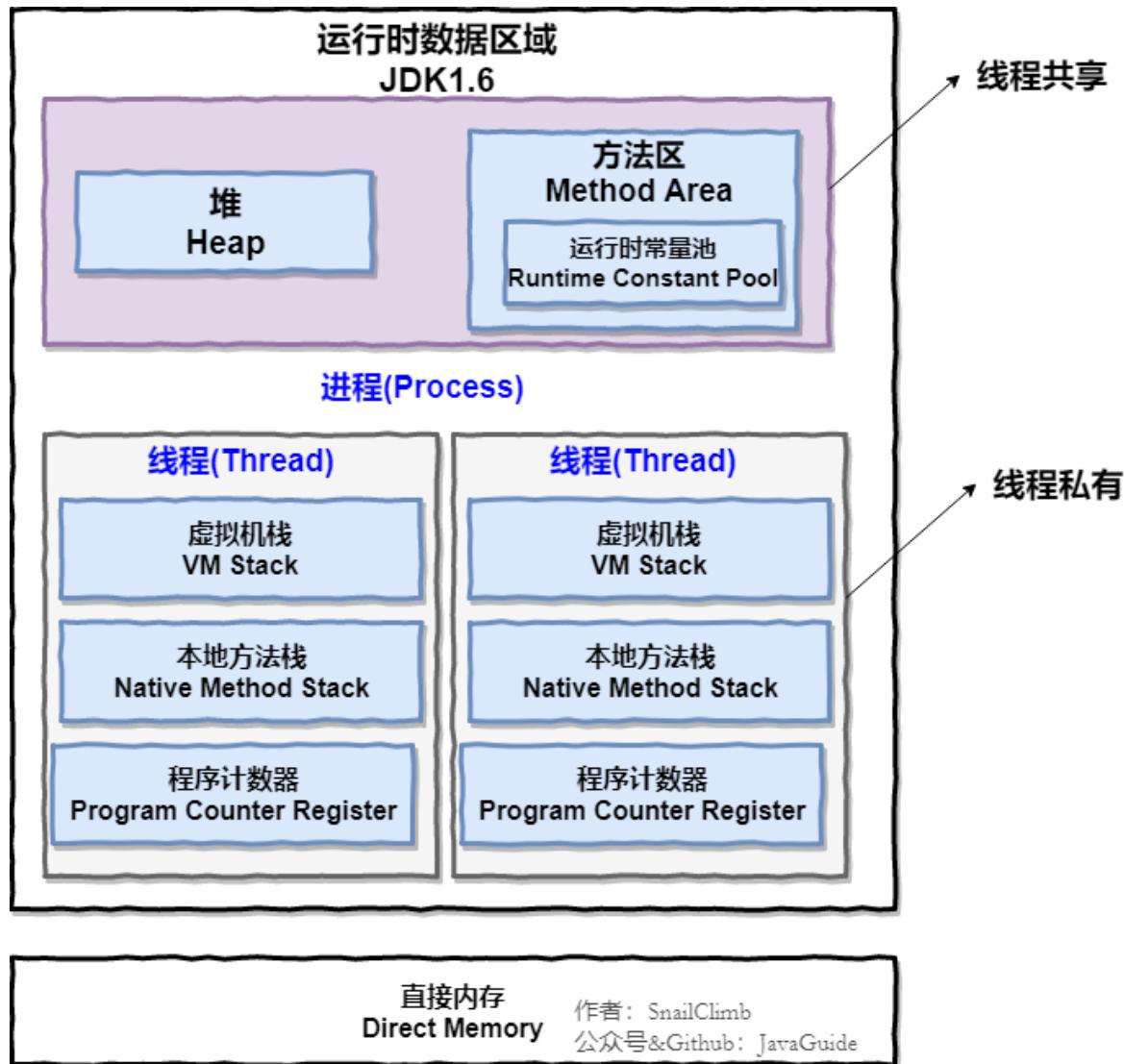
并且 JDK 1.8 的实现也在链表过长时会转换为红黑树。

# Java虚拟机

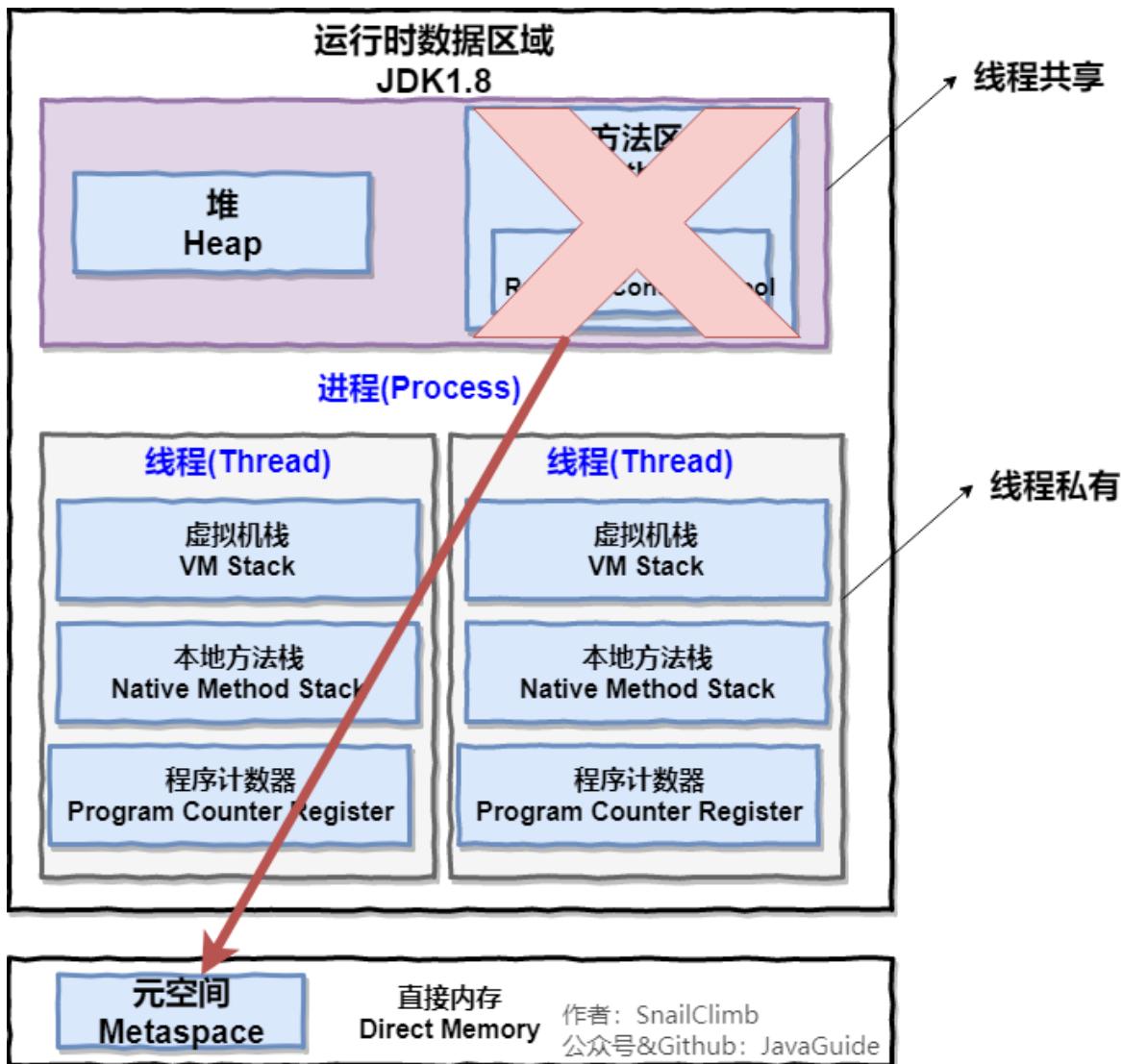
## 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK 1.8 和之前的版本略有不同，下面会介绍到。

JDK 1.8 之前：



JDK 1.8：



线程私有的：

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的：

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)

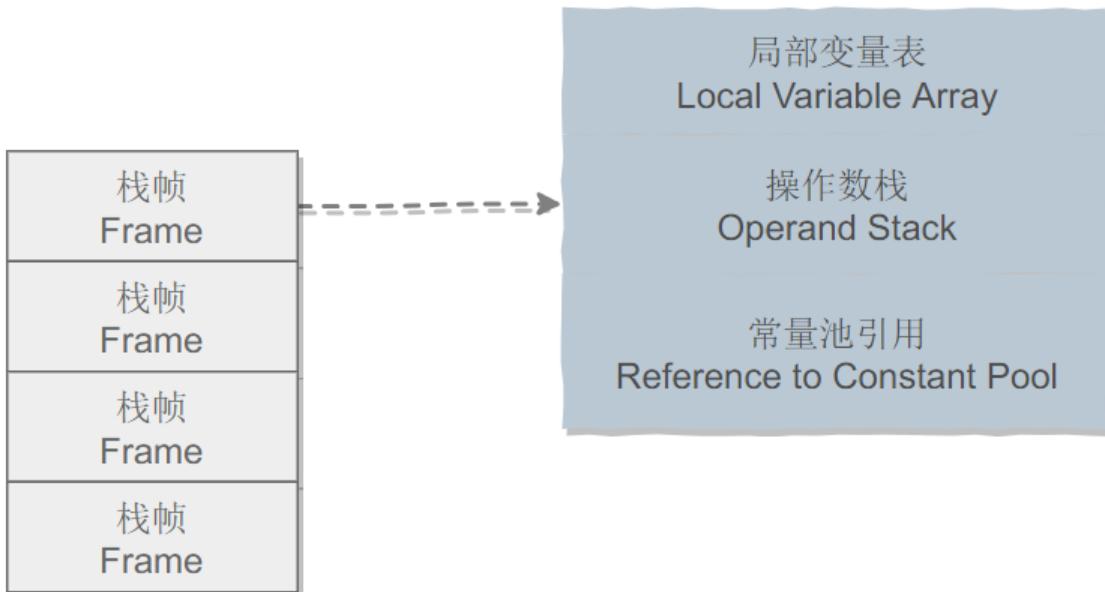
## 程序计数器

记录正在执行的虚拟机字节码指令的地址（如果正在执行的是本地方法则为空）。

**注意：程序计数器是唯一一个不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。**

## Java 虚拟机栈

每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。



CyC2018

- 局部变量表：包含方法执行过程中的所有变量
- 操作数栈：入栈、出栈、复制、交换、产生消费变量

可以通过 -Xss 这个虚拟机参数来指定每个线程的 Java 虚拟机栈内存大小：，

```
java -Xss512M HackTheJava
```

该区域可能抛出以下异常：

- 当线程请求的栈深度超过最大值，会抛出 StackOverflowError 异常；
- 栈进行动态扩展时如果无法申请到足够内存（虚拟机栈过多），会抛出 OutOfMemoryError 异常。

```
public void stackLeakByThread(){
    while(true){
        new Thread(){
            public void run(){
                while(true){

                }
            }
        }.start();
    }
}
```

## 本地方法栈

本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。

本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理。

## 堆

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC 堆（Garbage Collected Heap）**。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代（再细致一点有：Eden 空间、From Survivor、To Survivor 空间等）。**进一步划分的目的是更好地回收内存，或者更快地分配内存。**

eden 区、s0 区、s1 区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

## 方法区

用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

和堆一样不需要连续的内存，并且可以动态扩展，动态扩展失败一样会抛出 `OutOfMemoryError` 异常。

对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。

HotSpot 虚拟机把它当成永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 `OutOfMemoryError` 异常。为了更容易管理方法区，从 JDK 1.8 开始，移除永久代，并把方法区移至元空间，它位于本地内存中，而不是虚拟机内存中。

方法区是一个 JVM 规范，永久代与元空间都是其一种实现方式。在 JDK 1.8 之后，原来永久代的数据被分到了堆和元空间中。元空间存储类的元信息，静态变量和常量池等放入堆中。

### MetaSpace相比PermGen的优势

- 字符串常量池存在永久代中，容易出现性能问题和内存溢出
- 类和方法的信息大小难以确定，给永久代大小指定带来了困难
- 永久代会为GC带来不必要的复杂性
- HotSpot与其他JVM的集成

## 运行时常量池

运行时常量池是方法区的一部分。

Class 文件中的常量池（编译器生成的字面量和符号引用）会在类加载后被放入这个区域。

除了在编译期生成的常量，还允许动态生成，例如 String 类的 `intern()`。

## 直接内存

**直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 `OutOfMemoryError` 异常出现。**

在 JDK 1.4 中新引入了 NIO 类，它可以使 Native 函数库直接分配堆外内存，然后通过 Java 堆里的 `DirectByteBuffer` 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在堆内存和堆外内存来回拷贝数据。

### Java运行时内存中堆和栈的区别——内存分配策略

- 静态存储：编译时确定每个数据目标在运行时的存储空间需求
- 栈式存储：数据区需求在编译时未知，运行时模块入口前确定
- 堆式存储：编译时或运行时模块入口都无法确定，动态分配
- 联系：引用对象、数组时，栈里定义变量保存堆中目标的首地址
- 管理方式：栈自动释放，堆需要GC

- 空间大小：栈比堆小
- 碎片相关：栈产生的碎片远小于堆
- 分配方式：栈支持静态和动态分配，堆仅仅支持动态分配
- 效率：栈的效率比堆高

## 垃圾收集

垃圾收集主要是针对堆和方法区进行。程序计数器、虚拟机栈和本地方法栈这三个区域属于线程私有的，只存在于线程的生命周期内，线程结束之后就会消失，因此不需要对这三个区域进行垃圾回收。

### 判断一个对象是否可被回收

#### 1. 引用计数算法

为对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收。

在两个对象出现循环引用的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收。正是因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法。

```
public class Test {  
  
    public Object instance = null;  
  
    public static void main(String[] args) {  
        Test a = new Test();  
        Test b = new Test();  
        a.instance = b;  
        b.instance = a;  
        a = null;  
        b = null;  
        doSomething();  
    }  
}
```

在上述代码中，a 与 b 引用的对象实例互相持有了对象的引用，因此当我们把对 a 对象与 b 对象的引用去除之后，由于两个对象还存在互相之间的引用，导致两个 Test 对象无法被回收。

#### 2. 可达性分析算法

以 GC Roots 为起始点进行搜索，可达的对象都是存活的，不可达的对象可被回收。

Java 虚拟机使用该算法来判断对象是否可被回收，GC Roots 一般包含以下内容：

- 虚拟机栈中局部变量表中引用的对象
- 本地方法栈中 JNI (Native方法) 中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中的常量引用的对象

新生代用广度搜索，老生代用深度搜索

深度优先DFS一般采用递归方式实现，处理tracing的时候，可能会导致栈空间溢出，所以一般采用广度优先来实现tracing（递归情况下容易爆栈）。

广度优先的拷贝顺序使得GC后对象的空间局部性 (memory locality) 变差 (相关变量散开了)。

广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索算法法要快些。

深度优先搜索法占内存少但速度较慢，广度优先搜索算法占内存多但速度较快。

### 3. 方法区的回收

因为方法区主要存放永久代对象，而永久代对象的回收率比新生代低很多，所以在方法区上进行回收性价比不高。

主要是对常量池的回收和对类的卸载。

为了避免内存溢出，在大量使用反射和动态代理的场景都需要虚拟机具备类卸载功能。

类的卸载条件很多，需要满足以下三个条件，并且满足了条件也不一定会被卸载：

- 该类所有的实例都已经被回收，此时堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 Class 对象没有在任何地方被引用，也就无法在任何地方通过反射访问该类方法。

### 4. finalize()

类似 C++ 的析构函数，用于关闭外部资源。但是 try-finally 等方式可以做得更好，并且该方法运行代价很高，不确定性大，无法保证各个对象的调用顺序，因此最好不要使用。

当一个对象可被回收时，如果需要执行该对象的 finalize() 方法，那么就有可能在该方法中让对象重新被引用，从而实现自救。自救只能进行一次，如果回收的对象之前调用了 finalize() 方法自救，后面回收时不会再调用该方法。

## 5. 彻底死亡条件

条件1：通过GC Roots作为起点的向下搜索形成引用链，没有搜到该对象，这是第一次标记。

条件2：在finalize方法中没有逃脱回收（将自身被其他对象引用），这是第一次标记的清理。

## 引用类型

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

Java中提供这四种引用类型主要有两个目的：第一是可以让程序员通过代码的方式决定某些对象的生命周期；第二是有利于JVM进行垃圾回收。

### 1. 强引用

被强引用关联的对象不会被回收。

使用 new 一个新对象的方式来创建强引用。

```
Object obj = new Object();
```

如果想中断强引用和某个对象之间的关联，可以显示地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象。

### 2. 软引用

被软引用关联的对象只有在内存不够的情况下才会被回收。可以用来实现缓存：比如网页缓存、图片缓存等。

假如有一个应用需要读取大量的本地图片，如果每次读取图片都从硬盘读取，则会严重影响性能，但是如果全部加载到内存当中，又有可能造成内存溢出，此时使用软引用可以解决这个问题。

设计思路是：用一个HashMap来保存图片的路径 和 相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占用的空间，从而有效地避免了OOM的问题。在Android开发中对于大量图片下载会经常用到。

使用 SoftReference 类来创建软引用。

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null; // 使对象只被软引用关联
```

### 3. 弱引用

被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。

使用 WeakReference 类来创建弱引用。

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
```

### 4. 虚引用

又称为幽灵引用或者幻影引用，一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到一个对象。

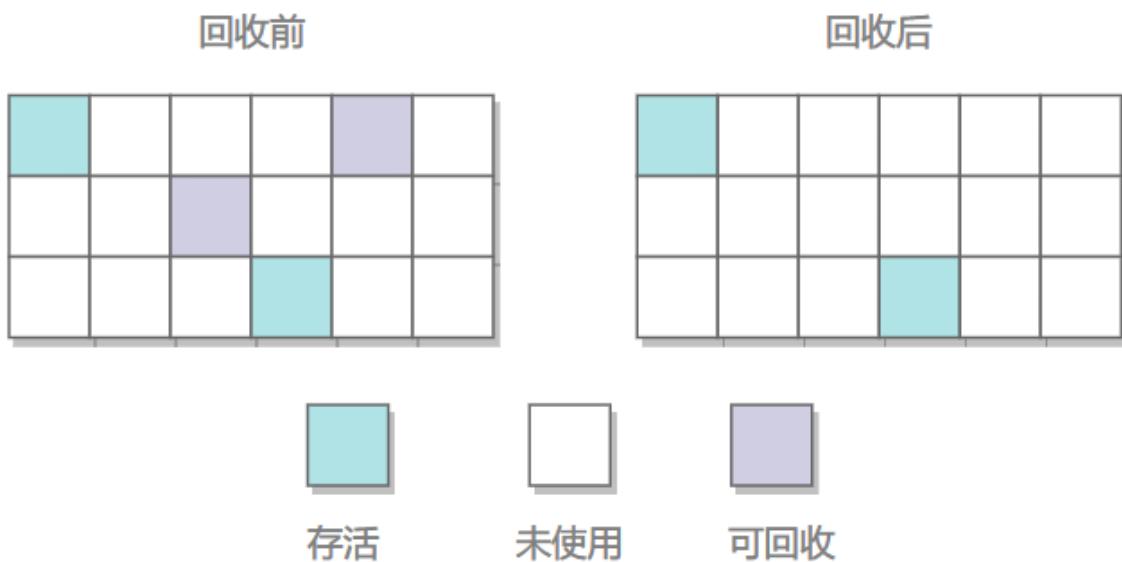
为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知，跟踪对象被垃圾收集器回收的活动，起哨兵作用。

使用 PhantomReference 来创建虚引用。

```
Object obj = new Object();
ReferenceQueue queue = new ReferenceQueue();
PhantomReference<Object> pf = new PhantomReference<Object>(obj, queue);
obj = null;
```

## 垃圾收集算法

### 1. 标记 - 清除



在标记阶段，程序会检查每个对象是否为活动对象，如果是活动对象，则程序会在对象头部打上标记。

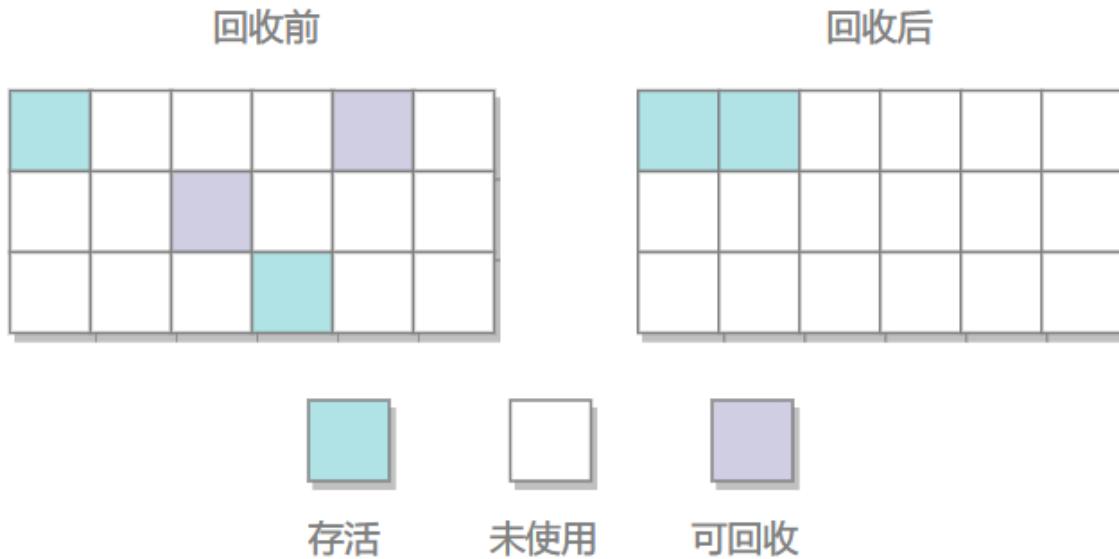
在清除阶段，会进行对象回收并取消标志位，另外，还会判断回收后的分块与前一个空闲分块是否连续，若连续，会合并这两个分块。回收对象就是把对象作为分块，连接到被称为“空闲链表”的单向链表，之后进行分配时只需要遍历这个空闲链表，就可以找到分块。

在分配时，程序会搜索空闲链表寻找空间大于等于新对象大小  $size$  的块  $block$ 。如果它找到的块等于  $size$ ，会直接返回这个分块；如果找到的块大于  $size$ ，会将块分割成大小为  $size$  与  $(block - size)$  的两部分，返回大小为  $size$  的分块，并把大小为  $(block - size)$  的块返回给空闲链表。

不足：

- 标记和清除过程效率都不高；
- 会产生大量不连续的内存碎片，导致无法给大对象分配内存。

## 2. 标记 - 整理



让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

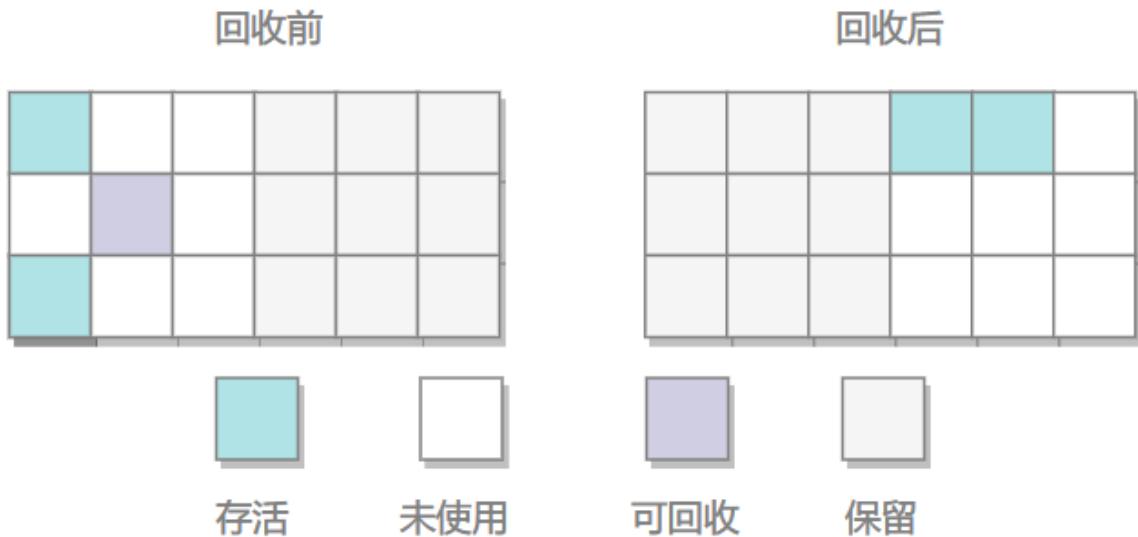
优点：

- 不会产生内存碎片

不足：

- 需要移动大量对象，处理效率比较低。

## 3. 复制



将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。

主要不足是只使用了内存的一半。

现在的商业虚拟机都采用这种收集算法回收新生代，但是并不是划分为大小相等的两块，而是一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。在回收时，将 Eden 和 Survivor 中还活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的那一块 Survivor。

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1:1，保证了内存的利用率达到 90%。如果每次回收有多于 10% 的对象存活，那么一块 Survivor 就不够用了，此时需要依赖于老年代进行空间分配担保，也就是借用老年代的空间存储放不下的对象。

## 4. 分代收集

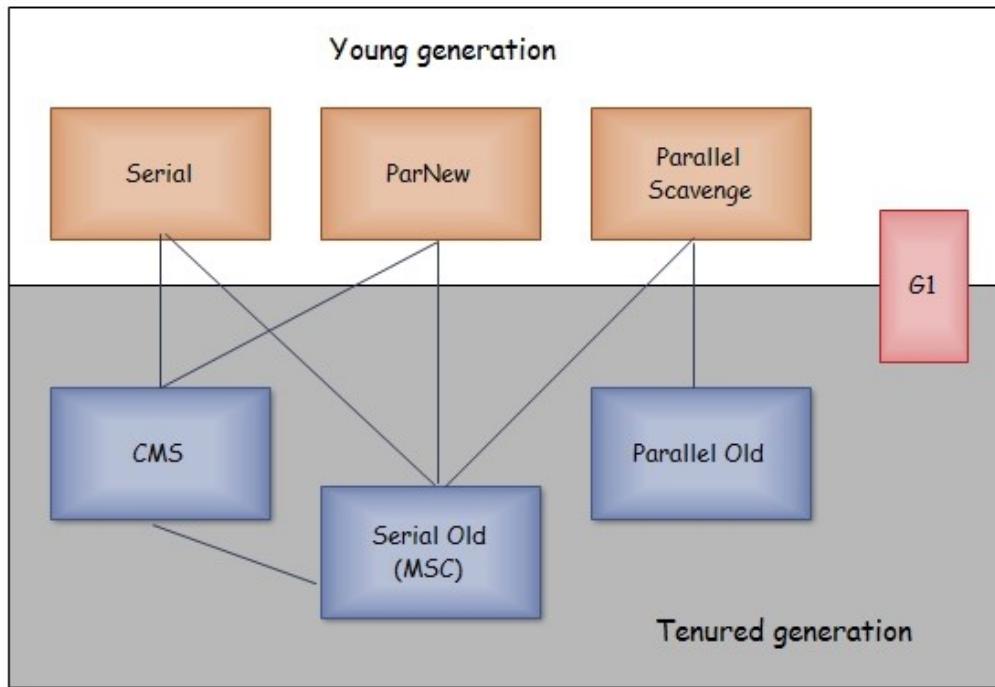
现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

一般将堆分为新生代和老年代。

- 新生代使用：复制算法
- 老年代使用：标记 - 清除 或者 标记 - 整理 算法

## 垃圾收集器

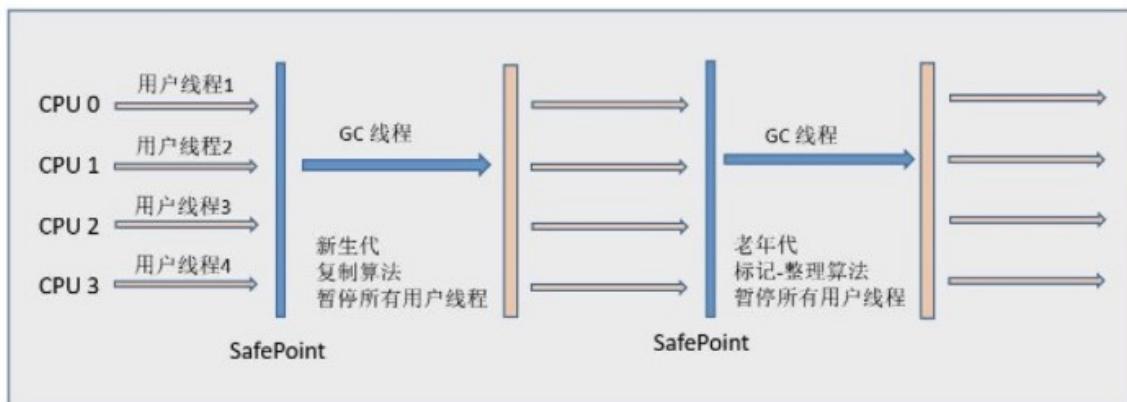
---



以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

- 单线程与多线程：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
- 串行与并行：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；并行指的是垃圾收集器和用户程序同时执行。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

## 1. Serial 收集器



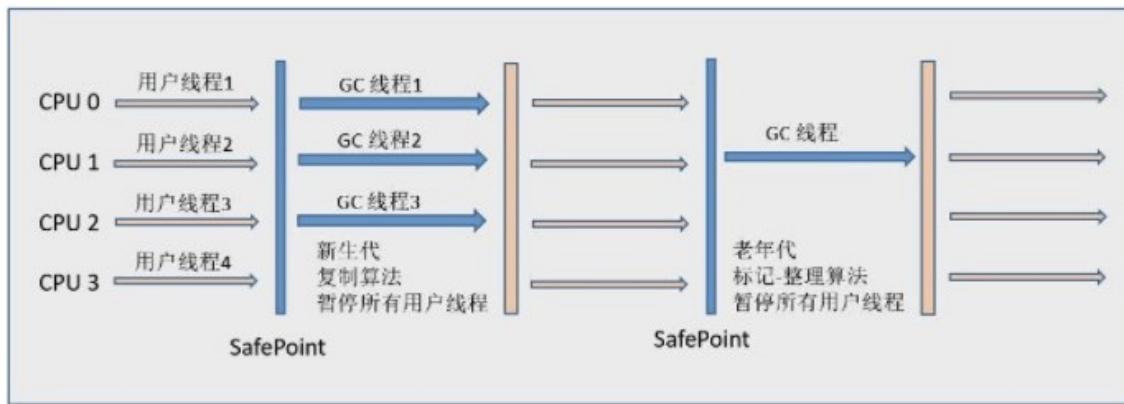
Serial 翻译为串行，也就是说它以串行的方式执行。

它是单线程的收集器，只会使用一个线程进行垃圾收集工作。

它的优点是简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

它是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

## 2. ParNew 收集器



它是 Serial 收集器的多线程版本。

它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用。

### 3. Parallel Scavenge 收集器

与 ParNew 一样是多线程收集器。

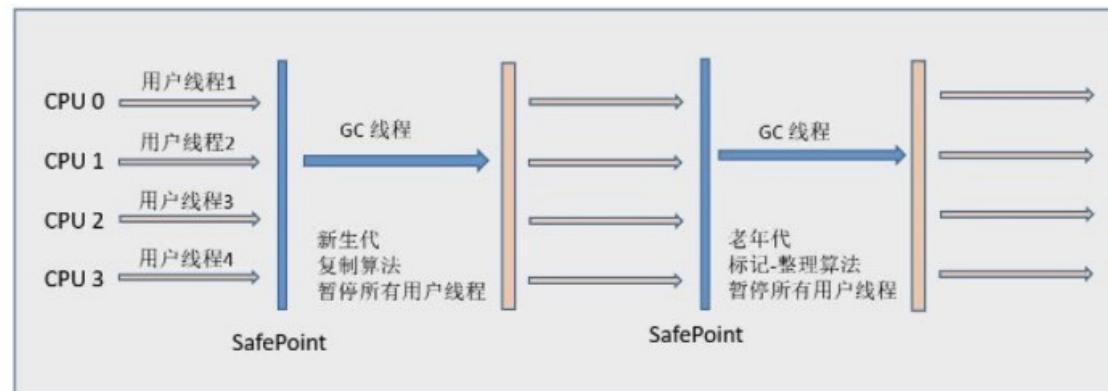
其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而它的目标是达到一个可控制的吞吐量，因此它被称为“吞吐量优先”收集器。这里的吞吐量指 CPU 用于运行用户程序的时间占总时间的比值。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。

缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。

可以通过一个开关参数打开 GC 自适应的调节策略 (GC Ergonomics)，就不需要手工指定新生代的大小 (-Xmn)、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

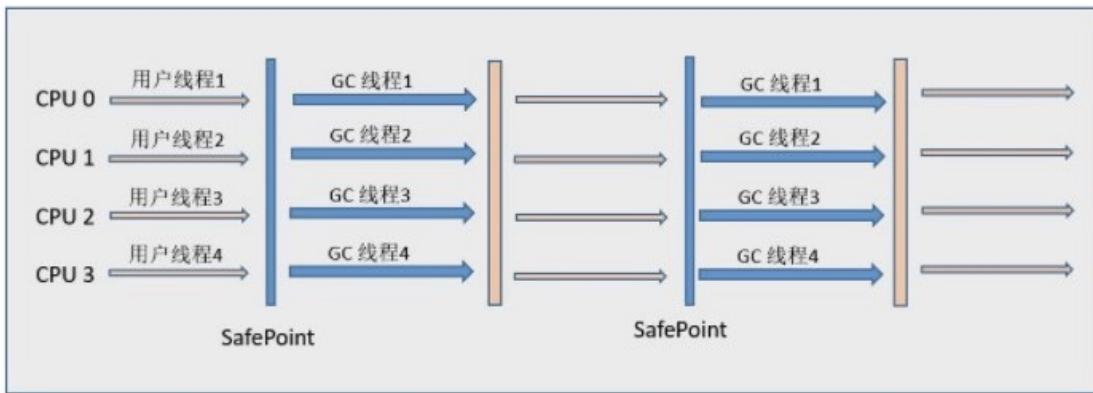
### 4. Serial Old 收集器



是 Serial 收集器的老年代版本，也是给 Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大用途：

- 在 JDK 1.5 以及之前版本 (Parallel Old 诞生以前) 中与 Parallel Scavenge 收集器搭配使用。
- 作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

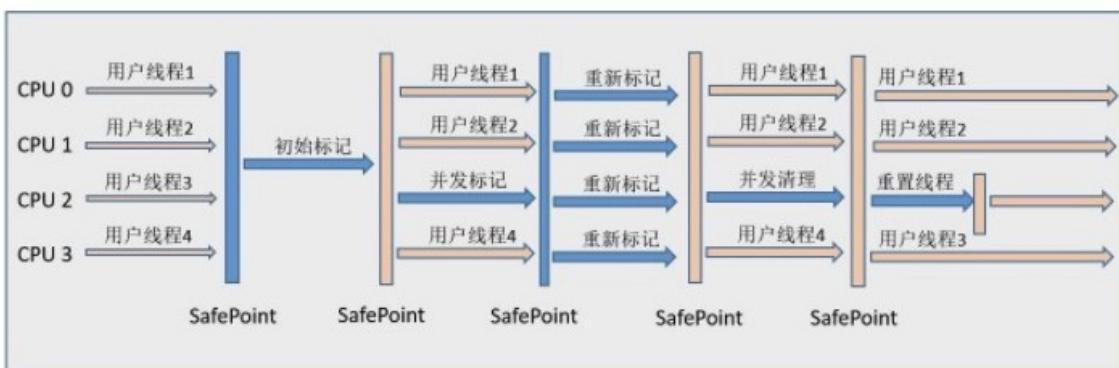
### 5. Parallel Old 收集器



是 Parallel Scavenge 收集器的老年代版本。

在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

## 6. CMS 收集器



CMS (Concurrent Mark Sweep) , Mark Sweep 指的是标记 - 清除算法。

分为以下四个流程：

- 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
- 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
- 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
- 并发清除：不需要停顿。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

具有以下缺点：

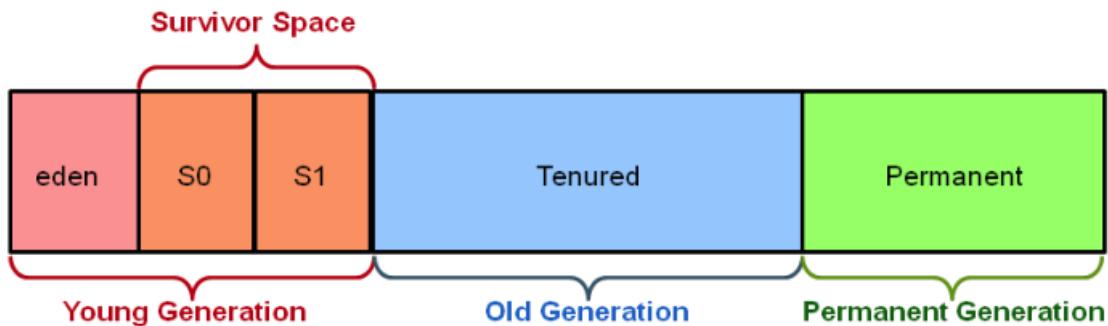
- 吞吐量低：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。
- 无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。
- 标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

## 7. G1 收集器

G1 (Garbage-First) ，它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

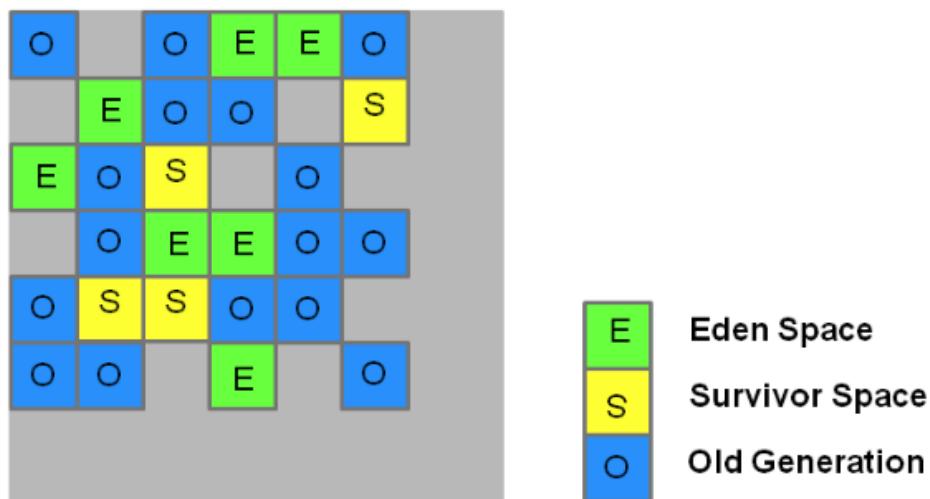
堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

## Hotspot Heap Structure



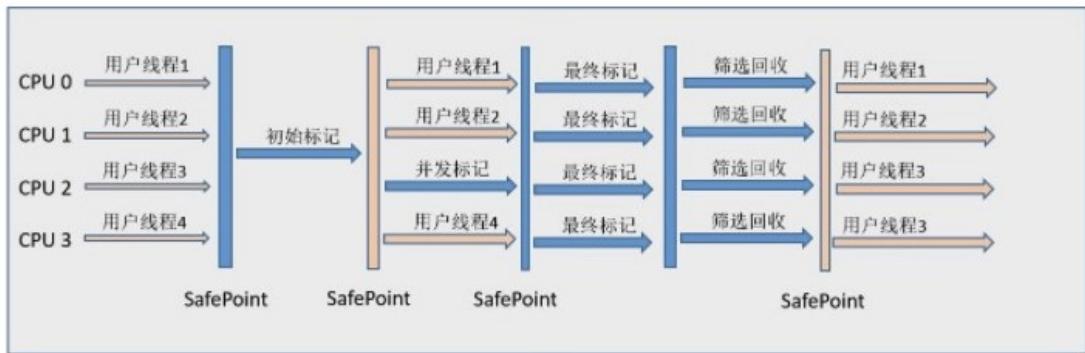
G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。

## G1 Heap Allocation



通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记：为了修正正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程，但是可并行执行。
- 筛选回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

具备如下特点：

- 空间整合：整体来看是基于“标记 - 整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，这意味着运行期间不会产生内存空间碎片。
- 可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒。

## 垃圾收集器比较

收集器	单线程/ 并行	串行/ 并发	新生代/老 年代	收集算法	目标	适用场景
<b>Serial</b>	单线程	串行	新生代	复制	响应速度优先	单 CPU 环境下的 Client 模式
<b>ParNew</b>	并行	串行	新生代	复制算法	响应速度优先	多 CPU 环境时在 Server 模式下与 CMS 配合
<b>Parallel Scavenge</b>	并行	串行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
<b>Serial Old</b>	单线程	串行	老年代	标记-整理	响应速度优先	单 CPU 环境下的 Client 模式、CMS 的后备预案
<b>Parallel Old</b>	并行	串行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务

CMS 收集器	单线程/并行	串行/并发	新生代/老年年代	标记-清除法	响应速度优先	集中在互联网站或 B/S 系统上的 Java 应用
		并发	新生代	标记-整理	响应速度优先	
G1	并行	并发	+ 老年代	+ 复制算法	响应速度优先	面向服务端应用，将来替换 CMS

## 内存分配与回收策略

### Minor GC 和 Full GC

- Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

### 内存分配策略

#### 1. 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

#### 2. 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

#### 3. 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

#### 4. 动态对象年龄判定

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

#### 5. 空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

### Full GC 的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

## 1. 调用 System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

## 2. 老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。

为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

## 3. 空间分配担保失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。

## 4. JDK 1.7 及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

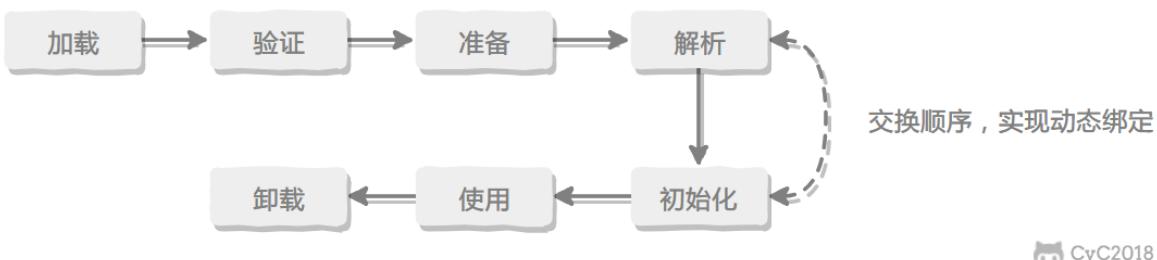
## 5. Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

## 类加载机制

类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。因为如果一次性加载，那么会占用很多的内存。

## 类的生命周期



包括以下 7 个阶段：

- 加载 (Loading)
- 验证 (Verification)
- 准备 (Preparation)
- 解析 (Resolution)
- 初始化 (Initialization)

- 使用 (Using)
- 卸载 (Unloading)

## 类加载过程

包含了加载、验证、准备、解析和初始化这 5 个阶段。

### 1. 加载

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

- 通过类的完全限定名称获取定义该类的二进制字节流。
- 将该字节流表示的静态存储结构转换为方法区的运行时存储结构。
- 在内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。

其中二进制字节流可以从以下方式中获取：

- 从 ZIP 包读取，成为 JAR、EAR、WAR 格式的基础。
- 从网络中获取，最典型的应用是 Applet。
- 运行时计算生成，例如动态代理技术，在 `java.lang.reflect.Proxy` 使用 `ProxyGenerator.generateProxyClass` 的代理类的二进制字节流。
- 由其他文件生成，例如由 JSP 文件生成对应的 Class 类。

### 2. 验证

确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

### 3. 准备

类变量是被 `static` 修饰的变量，准备阶段为类变量分配内存并设置初始值，使用的是方法区的内存。

实例变量不会在这阶段分配内存，它会在对象实例化时随着对象一起被分配在堆中。应该注意到，实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

初始值一般为 0 值，例如下面的类变量 `value` 被初始化为 0 而不是 123。

```
public static int value = 123;
```

如果类变量是常量，那么它将初始化为表达式所定义的值而不是 0。例如下面的常量 `value` 被初始化为 123 而不是 0。

```
public static final int value = 123;
```

### 4. 解析

将常量池的符号引用替换为直接引用的过程。

其中解析过程在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的动态绑定。

### 5. 初始化

初始化阶段才真正开始执行类中定义的 Java 程序代码。初始化阶段是虚拟机执行类构造器 () 方法的过程。在准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。

`()` 是由编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序由语句在源文件中出现的顺序决定。特别注意的是，静态语句块只能访问到定义在它之前的类变量，定义在它之后的类变量只能赋值，不能访问。例如以下代码：

```
public class Test {  
    static {  
        i = 0; // 给变量赋值可以正常编译通过  
        System.out.print(i); // 这句编译器会提示“非法向前引用”  
    }  
    static int i = 1;  
}
```

由于父类的 `()` 方法先执行，也就意味着父类中定义的静态语句块的执行要优先于子类。例如以下代码：

```
static class Parent {  
    public static int A = 1;  
    static {  
        A = 2;  
    }  
  
    static class Sub extends Parent {  
        public static int B = A;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(Sub.B); // 2  
    }  
}
```

接口中不可以使用静态语句块，但仍然有类变量初始化的赋值操作，因此接口与类一样都会生成 `()` 方法。但接口与类不同的是，执行接口的 `()` 方法不需要先执行父接口的 `()` 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的 `()` 方法。

虚拟机会保证一个类的 `()` 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 `()` 方法，其它线程都会阻塞等待，直到活动线程执行 `()` 方法完毕。如果在一个类的 `()` 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

## 类与类加载器

两个类相等，需要类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。

这里的相等，包括类的 `Class` 对象的 `equals()` 方法、`isAssignableFrom()` 方法、`isInstance()` 方法的返回结果为 `true`，也包括使用 `instanceof` 关键字做对象所属关系判定结果为 `true`。

## 类加载器分类

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

- 启动类加载器（Bootstrap ClassLoader），使用 C++ 实现，是虚拟机自身的一部分；
- 所有其它类的加载器，使用 Java 实现，独立于虚拟机，继承自抽象类 `java.lang.ClassLoader`。

从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

- 启动类加载器（Bootstrap ClassLoader）此类加载器负责将存放在 `<JRE_HOME>\lib` 目录中的，或者被 `-Xbootclasspath` 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如

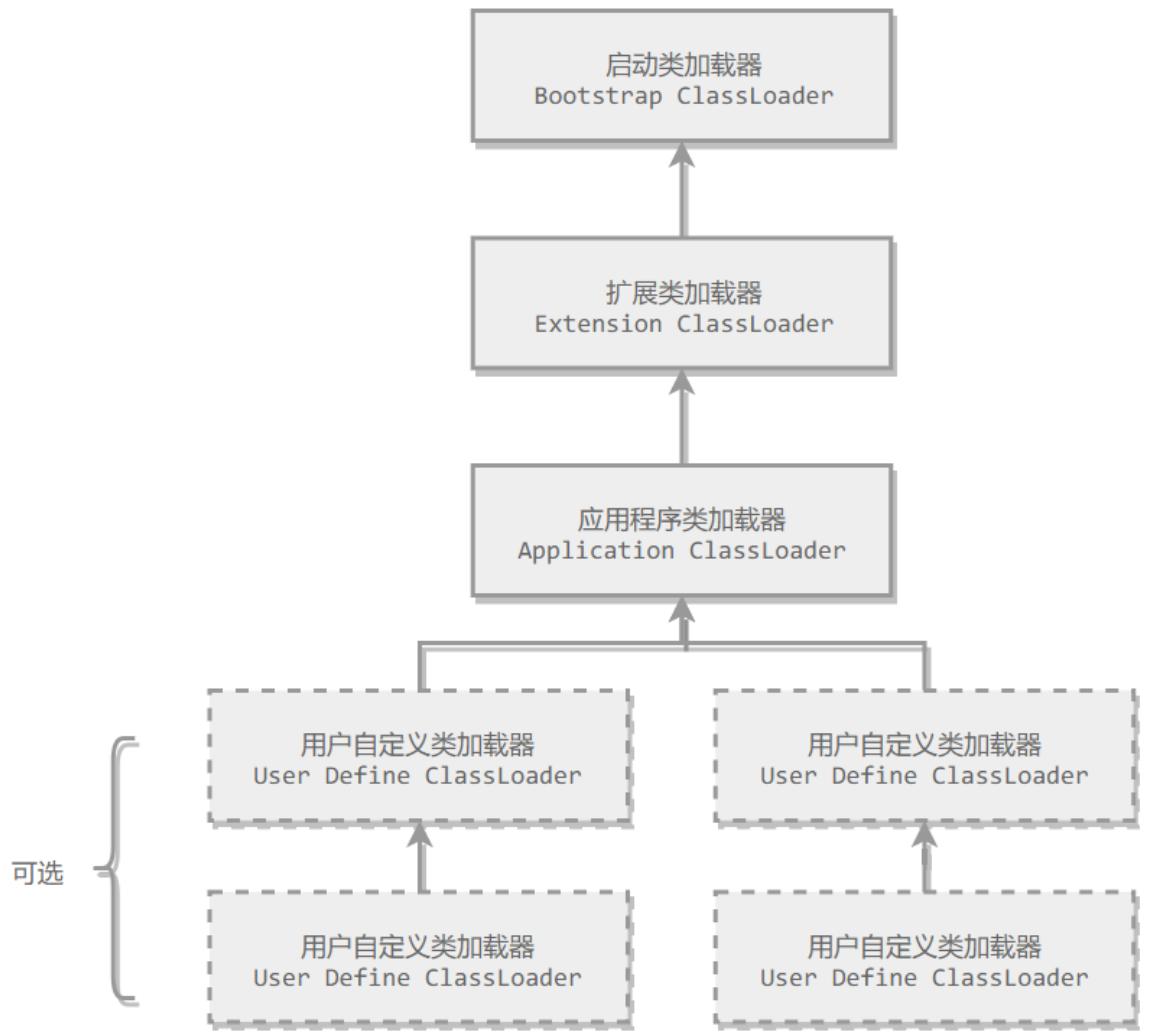
rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可。

- 扩展类加载器（Extension ClassLoader）这个类加载器是由 ExtClassLoader (sun.misc.Launcher\$ExtClassLoader) 实现的。它负责将 <JAVA\_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
- 应用程序类加载器（Application ClassLoader）这个类加载器是由 AppClassLoader (sun.misc.Launcher\$AppClassLoader) 实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

## 双亲委派模型

应用程序是由三种类加载器互相配合从而实现类加载，除此之外还可以加入自己定义的类加载器。

下图展示了类加载器之间的层次关系，称为双亲委派模型（Parents Delegation Model）。该模型要求除了顶层的启动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）。



## 1. 工作过程

一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。

## 2. 好处

使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。

例如 java.lang.Object 存放在 rt.jar 中，如果编写另外一个 java.lang.Object 并放到 ClassPath 中，程序可以编译通过。由于双亲委派模型的存在，所以在 rt.jar 中的 Object 比在 ClassPath 中的 Object 优先级更高，这是因为 rt.jar 中的 Object 使用的是启动类加载器，而 ClassPath 中的 Object 使用的是应用程序类加载器。rt.jar 中的 Object 优先级更高，那么程序中所有的 Object 都是这个 Object。

## 3. 实现

以下是抽象类 java.lang.ClassLoader 的代码片段，其中的 loadClass() 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 ClassNotFoundException，此时尝试自己去加载。

```
public abstract class ClassLoader {
    // The parent class loader for delegation
    private final ClassLoader parent;

    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

    protected Class<?> loadClass(String name, boolean resolve) throws
    ClassNotFoundException {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }
            }
            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
}
```

# 破坏双亲委派模型

双亲委派模型主要出现过3次较大的“被破坏”情况：

- 双亲委派模型出现之前——即JDK1.2发布之前

- 双亲委派模型在JDK1.2之后才被引入，类加载器和抽象类java.lang.ClassLoader在JDK1.0时代就已存在，面对已经存在的用户自定义类加载器的实现代码，Java设计者引入双亲委派模型时不得不做一些妥协。
- 考虑向前兼容，JDK1.2之后的java.lang.ClassLoader中添加了一个新的方法findClass()。
- JDK1.2之后不提倡用户覆盖loadClass()方法，而是把加载逻辑写到findClass()中，在loadClass()方法逻辑里如果父类加载失败，则调用findClass()方法加载，如此即符合双亲委派。

- 双亲委派模型自身缺陷

- 越基础的类越由上层加载器加载，若基础类需要调用回用户的代码，如JNI。
- 线程上下文加载器（Thread Context ClassLoader）。这个类加载器可以通过java.lang.Thread类的setContextClassLoader()方法进行设置。如果创建线程时还未设置，将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置，这个类加载器默认就是应用程序类加载器。

- 用户对程序动态性的追求导致

- OSGi环境下，类加载器不再是双亲委派模型的树状结构，而是进一步发展为网状结构。

## 自定义类加载器实现

以下代码中的 FileSystemClassLoader 是自定义类加载器，继承自 java.lang.ClassLoader，用于加载文件系统上的类。它首先根据类的全名在文件系统上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 defineClass() 方法来把这些字节代码转换成 java.lang.Class 类的实例。

java.lang.ClassLoader 的 loadClass() 实现了双亲委派模型的逻辑，自定义类加载器一般不去重写它，但是需要重写 findClass() 方法。

```
public class FileSystemClassLoader extends ClassLoader {  
  
    private String rootDir;  
  
    public FileSystemClassLoader(String rootDir) {  
        this.rootDir = rootDir;  
    }  
  
    protected Class<?> findClass(String name) throws ClassNotFoundException {  
        byte[] classData = getClassData(name);  
        if (classData == null) {  
            throw new ClassNotFoundException();  
        } else {  
            return defineClass(name, classData, 0, classData.length);  
        }  
    }  
  
    private byte[] getClassData(String className) {  
        String path = classNameToPath(className);  
        try {  
            InputStream ins = new FileInputStream(path);  
            ByteArrayOutputStream baos = new ByteArrayOutputStream();  
            int bufferSize = 4096;  
            byte[] buffer = new byte[bufferSize];  
            int len;  
            while ((len = ins.read(buffer)) != -1) {  
                baos.write(buffer, 0, len);  
            }  
            ins.close();  
            return baos.toByteArray();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

```

        int bytesNumRead;
        while ((bytesNumRead = ins.read(buffer)) != -1) {
            baos.write(buffer, 0, bytesNumRead);
        }
        return baos.toByteArray();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

private String classNameToPath(String className) {
    return rootDir + File.separatorChar
        + className.replace('.', File.separatorChar) + ".class";
}
}

```

## 对象创建过程

### Java 创建对象的过程



微信公众号：Java面试通关手册

#### Step1:类加载检查

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

#### Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

**内存分配的两种方式：**（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的。



- 指针碰撞

- 适用场合：堆内存规整（即没有垃圾碎片）的情况下
- 原理：用过的内存全部整合到一边，没有用过的内存放到一边，中间有一个分界值指针，只需要想着没用过的内存方向将该指针移动对象内存大小位置即可
- GC收集器：Serial、ParNew
- 空闲列表
  - 适用场合：堆内存不规整的情况下
  - 原理：虚拟机会维护一个列表，该列表会记录哪些存储块是可用的，在分配的时候，找一块足够大的内存块划分给对象实例，最后更新列表记录
  - GC收集器：CMS

### **内存分配并发问题（补充内容，需要掌握）**

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**
- **TLAB：** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

### **Step3: 初始化零值**

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

### **Step4: 设置对象头**

初始化零值完成之后，**虚拟机要对对象进行必要的设置**，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。**这些信息存放在对象头中**。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

### **Step5: 执行 init 方法**

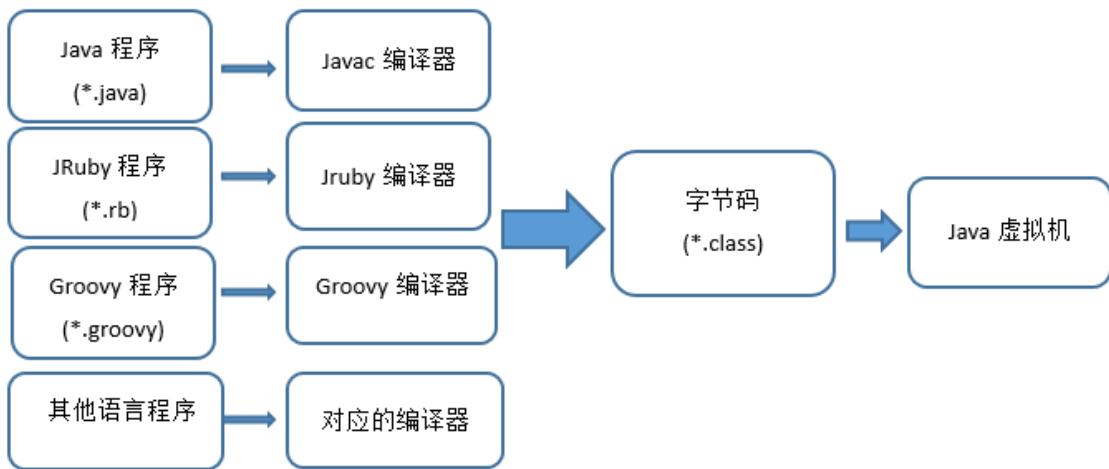
在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

## **字节码**

---

在 Java 中，JVM 可以理解的代码就叫做 **字节码**（即扩展名为 `.class` 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Clojure (Lisp 语言的一种方言)、Groovy、Scala 等语言都是运行在 Java 虚拟机之上。下图展示了不同的语言被不同的编译器编译成 `.class` 文件最终运行在 Java 虚拟机之上。`.class` 文件的二进制格式可以使用 [WinHex](#) 查看。



可以说.class文件是不同的语言在 Java 虚拟机之间的重要桥梁，同时也是支持 Java 跨平台很重要的一个原因。

## JDK 监控和故障处理工具总结

### JDK 命令行工具

这些命令在 JDK 安装目录下的 bin 目录下：

- **jps** (JVM Process Status) : 类似 UNIX 的 `ps` 命令。用户查看所有 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息;
- **jstat** ( JVM Statistics Monitoring Tool) : 用于收集 HotSpot 虚拟机各方面的运行数据;
- **jinfo** (Configuration Info for Java) : Configuration Info for Java, 显示虚拟机配置信息;
- **jmap** (Memory Map for Java) : 生成堆转储快照;
- **jhat** (JVM Heap Dump Browser) : 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果;
- **jstack** (Stack Trace for Java): 生成虚拟机当前时刻的线程快照，线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

### `jps`: 查看所有 Java 进程

`jps` (JVM Process Status) 命令类似 UNIX 的 `ps` 命令。

`jps`：显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID (Local Virtual Machine Identifier,LVMIID)。`jps -q`：只输出进程的本地虚拟机唯一 ID。

```
C:\Users\Snailclimb>jps
7360 NettyClient2
17396
7972 Launcher
16504 Jps
17340 NettyServer
```

`jps -l`：输出主类的全名，如果进程执行的是 Jar 包，输出 Jar 路径。

```
C:\Users\SnailClimb>jps -l
7360 firstNettyDemo.NettyClient2
17396
7972 org.jetbrains.jps.cmdline.Launcher
16492 sun.tools.jps.Jps
17340 firstNettyDemo.NettyServer
```

jps -v : 输出虚拟机进程启动时 JVM 参数。

jps -m : 输出传递给 Java 进程 main() 函数的参数。

## jstat: 监视虚拟机各种运行状态信息

jstat (JVM Statistics Monitoring Tool) 使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程（需要远程主机提供 RMI 支持）虚拟机进程中的类信息、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。

**jstat 命令使用格式：**

```
jstat [<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

比如 jstat -gc -h3 31736 1000 10 表示分析进程 id 为 31736 的 gc 情况，每隔 1000ms 打印一次记录，打印 10 次停止，每 3 行后打印指标头部。

**常见的 option 如下：**

- jstat -class vmid : 显示 ClassLoader 的相关信息；
- jstat -compiler vmid : 显示 JIT 编译的相关信息；
- jstat -gc vmid : 显示与 GC 相关的堆信息；
- jstat -gccapacity vmid : 显示各个代的容量及使用情况；
- jstat -gcnew vmid : 显示新生代信息；
- jstat -gcnewcapacity vmid : 显示新生代大小与使用情况；
- jstat -gcold vmid : 显示老年代和永久代的信息；
- jstat -gcoldcapacity vmid : 显示老年代的大小；
- jstat -gcpermcapacity vmid : 显示永久代大小；
- jstat -gcutil vmid : 显示垃圾收集信息；

另外，加上 -t 参数可以在输出信息上加一个 Timestamp 列，显示程序的运行时间。

## jinfo: 实时地查看和调整虚拟机各项参数

jinfo vmid : 输出当前 jvm 进程的全部参数和系统属性 (第一部分是系统的属性，第二部分是 JVM 的参数)。

jinfo -flag name vmid : 输出对应名称的参数的具体值。比如输出 MaxHeapSize、查看当前 jvm 进程是否开启打印 GC 日志 (-XX:PrintGCDetails : 详细 GC 日志模式，这两个都是默认关闭的)。

```
C:\Users\SnailClimb>jinfo -flag MaxHeapSize 17340
-XX:MaxHeapSize=2124414976
```

```
C:\Users\SnailClimb>jinfo -flag PrintGC 17340
-XX:-PrintGC
```

使用 `jinfo` 可以在不重启虚拟机的情况下，可以动态的修改 jvm 的参数。尤其在线上的环境特别有用，请看下面的例子：

`jinfo -flag [+|-]name vmid` 开启或者关闭对应名称的参数。

```
C:\Users\Snailclimb>jinfo -flag PrintGC 17340  
-XX:-PrintGC  
  
C:\Users\Snailclimb>jinfo -flag +PrintGC 17340  
  
C:\Users\Snailclimb>jinfo -flag PrintGC 17340  
-XX:+PrintGC
```

## jmap:生成堆转储快照

`jmap` (Memory Map for Java) 命令用于生成堆转储快照。如果不使用 `jmap` 命令，要想获取 Java 堆转储，可以使用 “`-XX:+HeapDumpOnOutOfMemoryError`” 参数，可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，Linux 命令下可以通过 `kill -3` 发送进程退出信号也能拿到 dump 文件。

`jmap` 的作用并不仅仅是为了获取 dump 文件，它还可以查询 finalizer 执行队列、Java 堆和永久代的详细信息，如空间使用率、当前使用的是哪种收集器等。和 `jinfo` 一样，`jmap` 有不少功能在 Windows 平台下也是受限制的。

示例：将指定应用程序的堆快照输出到桌面。后面，可以通过 `jhat`、`Visual VM` 等工具分析该堆文件。

```
C:\Users\Snailclimb>jmap -  
dump:format=b,file=C:\Users\Snailclimb\Desktop\heap.hprof 17340  
Dumping heap to C:\Users\Snailclimb\Desktop\heap.hprof ...  
Heap dump file created
```

## jhat: 分析 heapdump 文件

`jhat` 用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果。

```
C:\Users\Snailclimb>jhat C:\Users\Snailclimb\Desktop\heap.hprof  
Reading from C:\Users\Snailclimb\Desktop\heap.hprof...  
Dump file created Sat May 04 12:30:31 CST 2019  
Snapshot read, resolving...  
Resolving 131419 objects...  
Chasing references, expect 26 dots.....  
Eliminating duplicate references.....  
Snapshot resolved.  
Started HTTP server on port 7000  
Server is ready.
```

访问 <http://localhost:7000/>

## jstack :生成虚拟机当前时刻的线程快照

`jstack` (Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合。

生成线程快照的主要目的是定位线程长时间出现停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者在等待些什么资源。

下面是一个线程死锁的代码。我们下面会通过 jstack 命令进行死锁检查，输出死锁信息，找到发生死锁的线程。

```
public class DeadLockDemo {  
    private static Object resource1 = new Object() //资源 1  
    private static Object resource2 = new Object() //资源 2  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            synchronized (resource1) {  
                System.out.println(Thread.currentThread() + "get resource1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + "waiting get  
resource2");  
                synchronized (resource2) {  
                    System.out.println(Thread.currentThread() + "get  
resource2");  
                }  
            }  
        }, "线程 1").start();  
  
        new Thread(() -> {  
            synchronized (resource2) {  
                System.out.println(Thread.currentThread() + "get resource2");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println(Thread.currentThread() + "waiting get  
resource1");  
                synchronized (resource1) {  
                    System.out.println(Thread.currentThread() + "get  
resource1");  
                }  
            }  
        }, "线程 2").start();  
    }  
}
```

Output

```
Thread[线程 1,5,main]get resource1  
Thread[线程 2,5,main]get resource2  
Thread[线程 1,5,main]waiting get resource2  
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000); 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也产生了死锁。

通过 jstack 命令分析：

```
C:\Users\SnailClimb>jps
13792 KotlinCompileDaemon
7360 NettyClient2
17396
7972 Launcher
8932 Launcher
9256 DeadLockDemo
10764 Jps
17340 NettyServer

C:\Users\SnailClimb>jstack 9256
```

输出的部分内容如下：

```
Found one Java-level deadlock:
=====
"线程 2":
  waiting to lock monitor 0x000000000333e668 (object 0x00000000d5efe1c0, a
  java.lang.Object),
  which is held by "线程 1"
"线程 1":
  waiting to lock monitor 0x000000000333be88 (object 0x00000000d5efe1d0, a
  java.lang.Object),
  which is held by "线程 2"

Java stack information for the threads listed above:
=====
"线程 2":
  at DeadLockDemo.lambda$main$1(DeadLockDemo.java:31)
  - waiting to lock <0x00000000d5efe1c0> (a java.lang.Object)
  - locked <0x00000000d5efe1d0> (a java.lang.Object)
  at DeadLockDemo$$Lambda$2/1078694789.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:748)

"线程 1":
  at DeadLockDemo.lambda$main$0(DeadLockDemo.java:16)
  - waiting to lock <0x00000000d5efe1d0> (a java.lang.Object)
  - locked <0x00000000d5efe1c0> (a java.lang.Object)
  at DeadLockDemo$$Lambda$1/1324119927.run(Unknown Source)
  at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

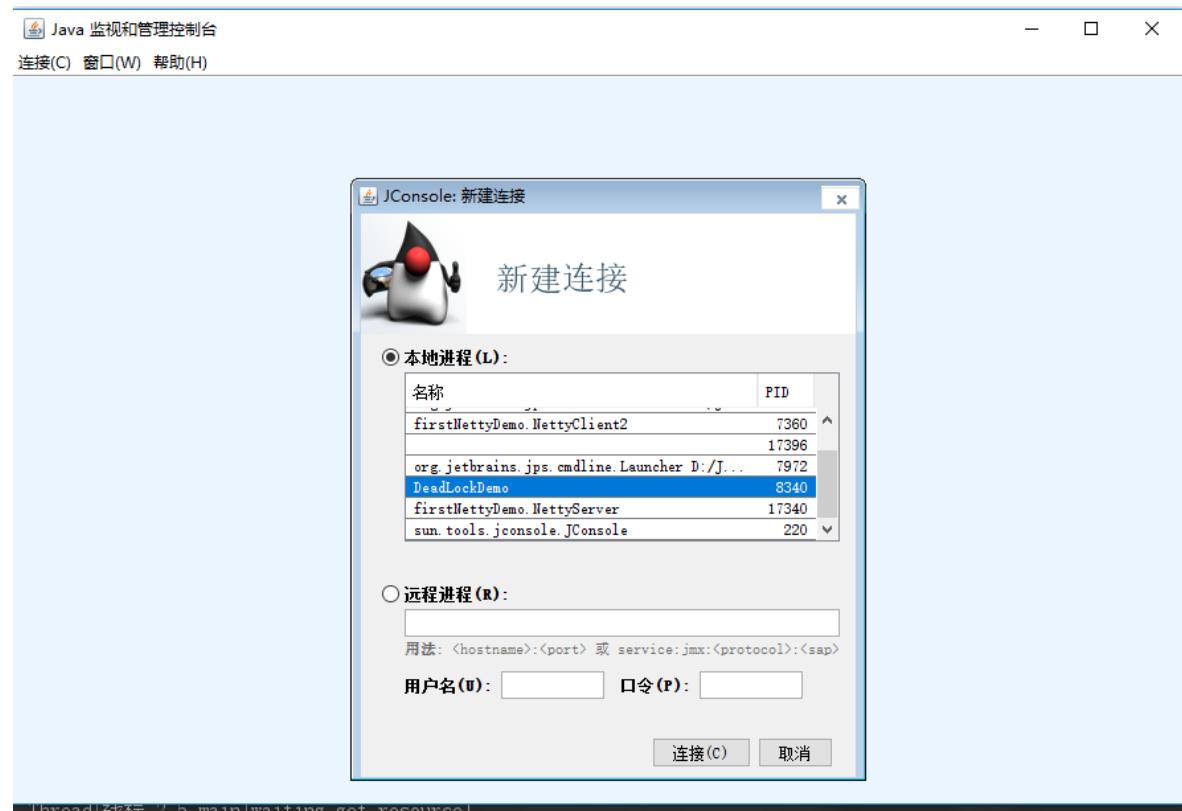
可以看到 jstack 命令已经帮我们找到发生死锁的线程的具体信息。

## JDK 可视化分析工具

### JConsole:Java 监视与管理控制台

JConsole 是基于 JMX 的可视化监视、管理工具。可以很方便的监视本地及远程服务器的 java 进程的内存使用情况。你可以在控制台输出 console 命令启动或者在 JDK 目录下的 bin 目录找到 jconsole.exe 然后双击启动。

## 连接 Jconsole



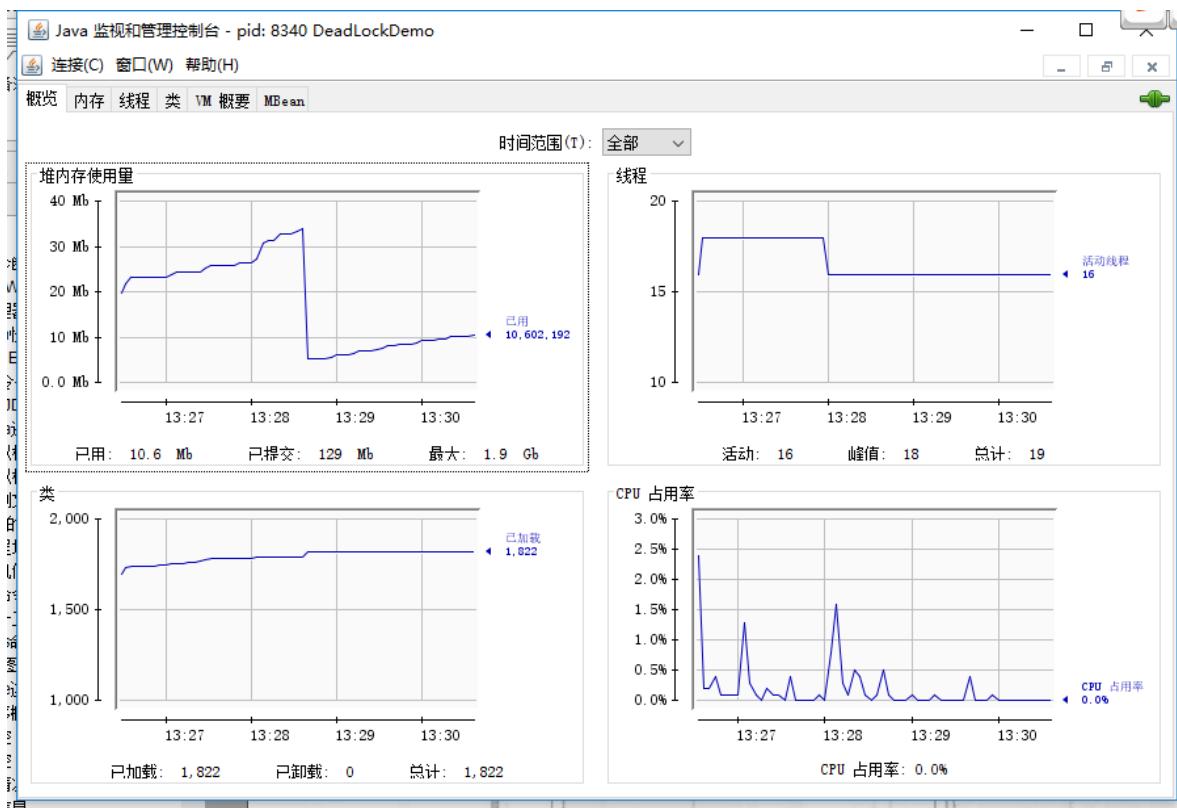
如果需要使用 JConsole 连接远程进程，可以在远程 Java 程序启动时加上下面这些参数：

```
-Djava.rmi.server.hostname=外网访问 ip 地址  
-Dcom.sun.management.jmxremote.port=60001 //监控的端口号  
-Dcom.sun.management.jmxremote.authenticate=false //关闭认证  
-Dcom.sun.management.jmxremote.ssl=false
```

在使用 JConsole 连接时，远程进程地址如下：

```
外网访问 ip 地址:60001
```

## 查看 Java 程序概况

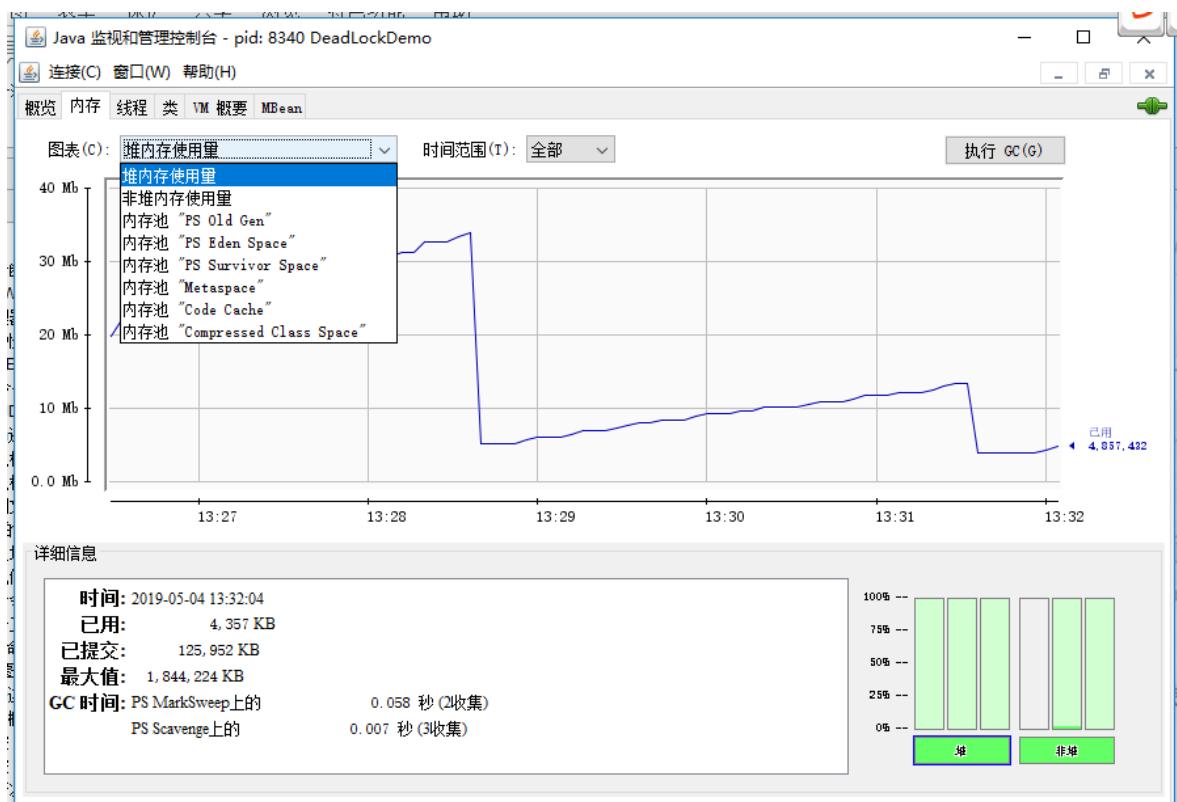


## 内存监控

JConsole 可以显示当前内存的详细信息。不仅包括堆内存/非堆内存的整体信息，还可以细化到 eden 区、survivor 区等的使用情况，如下图所示。

点击右边的“执行 GC(G)”按钮可以强制应用程序执行一个 Full GC。

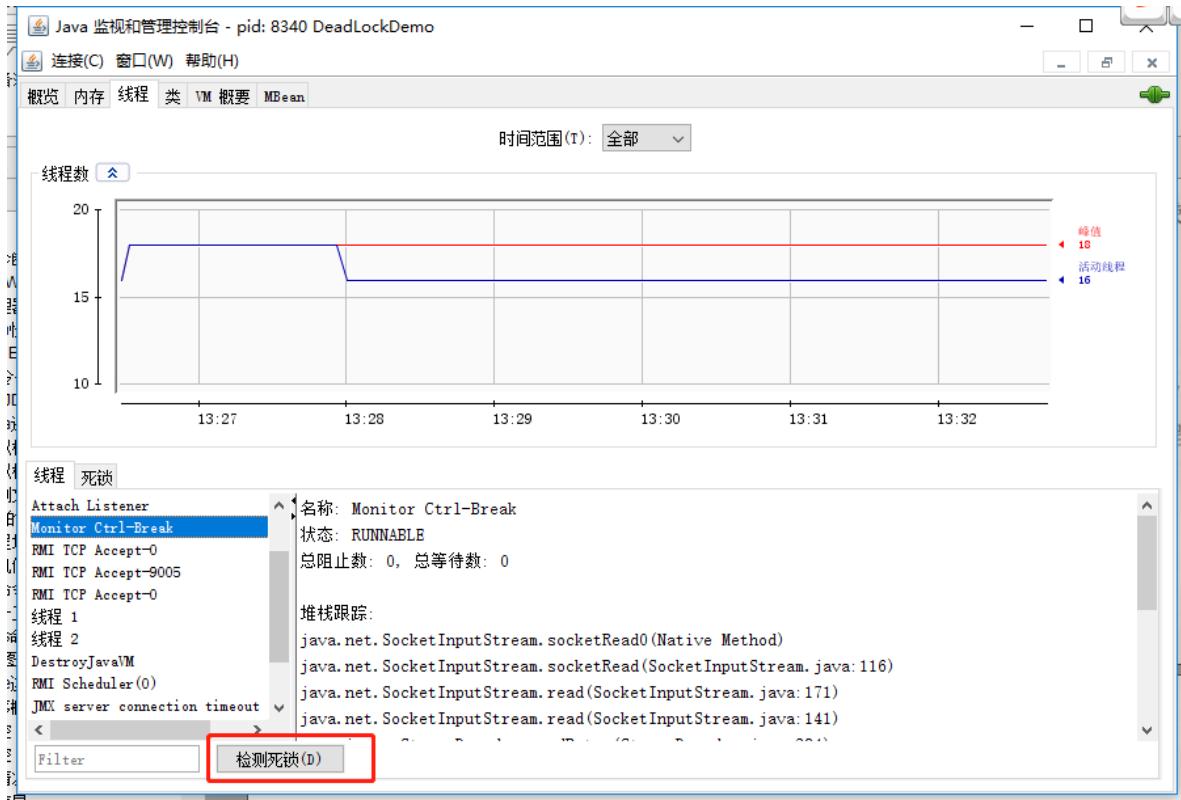
- 新生代 GC (Minor GC)** :指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- 老年代 GC (Major GC/Full GC)** :指发生在老年代的 GC，出现了 Major GC 经常会伴随着一次的 Minor GC (并非绝对)，Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。



## 线程监控

类似我们前面讲的 `jstack` 命令，不过这个是可视化的。

最下面有一个“检测死锁(D)”按钮，点击这个按钮可以自动为你找到发生死锁的线程以及它们的详细信息。



## Visual VM:多合一故障处理工具

VisualVM 提供在 Java 虚拟机 (Java Virtual Machine, JVM) 上运行的 Java 应用程序的详细信息。在 VisualVM 的图形用户界面中，您可以方便、快捷地查看多个 Java 应用程序的相关信息。Visual VM 官网：<https://visualvm.github.io/>。Visual VM 中文文档：<https://visualvm.github.io/documentation.html>。

下面这段话摘自《深入理解 Java 虚拟机》。

VisualVM (All-in-One Java Troubleshooting Tool) 是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序，官方在 VisualVM 的软件说明中写上了“All-in-One”的描述字样，预示着他除了运行监视、故障处理外，还提供了很多其他方面的功能，如性能分析 (Profiling)。VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少，而且 VisualVM 还有一个很大的优点：不需要被监视的程序基于特殊 Agent 运行，因此他对应用程序的实际性能的影响很小，使得他可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

VisualVM 基于 NetBeans 平台开发，因此他一开始就具备了插件扩展功能的特性，通过插件扩展支持，VisualVM 可以做到：

- 显示虚拟机进程以及进程的配置、环境信息 (`jps`、`jinfo`)。
- 监视应用程序的 CPU、GC、堆、方法区以及线程的信息 (`jstat`、`jstack`)。
- dump 以及分析堆转储快照 (`jmap`、`jhat`)。
- 方法级别的程序运行性能分析，找到被调用最多、运行时间最长的方法。
- 离线程序快照：收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照，可以将快照发送开发者处进行 Bug 反馈。
- 其他 plugins 的无限的可能性.....

这里就不具体介绍 VisualVM 的使用，如果想了解的话可以看：

- <https://visualvm.github.io/documentation.html>
- <https://www.ibm.com/developerworks/cn/java/j-lo-visualvm/index.html>

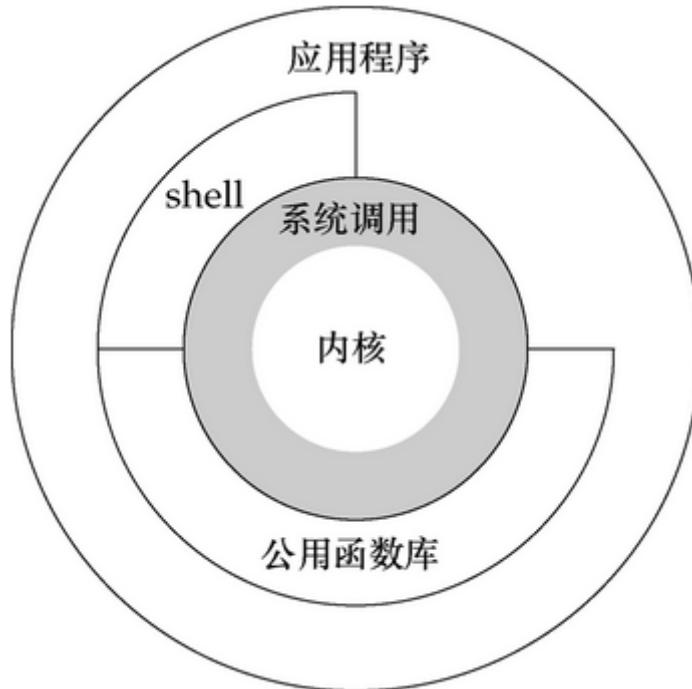
## 系统CPU飙高和GC频繁，如何排查

---

- 通过 `top` 命令查看CPU情况，如果CPU比较高，则通过 `top -Hp <pid>` 命令查看当前进程的各个线程运行情况，找出CPU过高的线程之后，将其线程id转换为十六进制的表现形式，然后在 `jstack` 日志中查看该线程主要在进行的工作。这里又分为两种情况
- 如果是正常的用户线程，则通过该线程的堆栈信息查看其具体是在哪处用户代码处运行比较消耗CPU；
- 如果该线程是 `VM Thread`，则通过 `jstat -gcutil <pid> <period> <times>` 命令监控当前系统的GC状况，然后通过 `jmap dump:format=b,file=<filepath> <pid>` 导出系统当前的内存数据。导出之后将内存情况放到eclipse的mat工具中进行分析即可得出内存中主要是什么对象比较消耗内存，进而可以处理相关代码；
- 如果通过 `top` 命令看到CPU并不高，并且系统内存占用率也比较低。此时就可以考虑是否是由于另外三种情况导致的问题。具体的可以根据具体情况分析：
- 如果是接口调用比较耗时，并且是不定时出现，则可以通过压测的方式加大阻塞点出现的频率，从而通过 `jstack` 查看堆栈信息，找到阻塞点；
- 如果是某个功能突然出现停滞的状况，这种情况也无法复现，此时可以通过多次导出 `jstack` 日志的方式对比哪些用户线程是一直都处于等待状态，这些线程就是可能存在问题是的线程；
- 如果通过 `jstack` 可以查看到死锁状态，则可以检查产生死锁的两个线程的具体阻塞点，从而处理相应的问题。

# Linux

## 用户态和内核态



## 概念

Linux的设计哲学之一就是：对不同的操作赋予不同的执行等级，就是所谓特权的概念，即与系统相关的一些特别关键的操作必须由最高特权的程序来完成。

Intel的X86架构的CPU提供了0到3四个特权级，数字越小，特权越高，Linux操作系统中主要采用了0和3两个特权级，分别对应的就是**内核态(Kernel Mode)**与**用户态(User Mode)**。

- 内核态：CPU可以访问内存所有数据，包括外围设备（硬盘、网卡），CPU也可以将自己从一个程序切换到另一个程序；
- 用户态：只能受限的访问内存，且不允许访问外围设备，占用CPU的能力被剥夺，CPU资源可以被其他程序获取；

Linux中任何一个用户进程被创建时都包含2个栈：内核栈，用户栈，并且是进程私有的，从用户态开始运行。内核态和用户态分别对应内核空间与用户空间，内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。

## 内核空间相关

- 内核空间：存放的是内核代码和数据，处于虚拟空间；
- 内核态：当进程执行系统调用而进入内核代码中执行时，称进程处于内核态，此时CPU处于特权级最高的0级内核代码中执行，当进程处于内核态时，执行的内核代码会使用当前进程的内核栈，每个进程都有自己的内核栈；
- CPU堆栈指针寄存器指向：内核栈地址；
- 内核栈：进程处于内核态时使用的栈，存在于内核空间；
- 处于内核态进程的权利：处于内核态的进程，当它占有CPU的时候，可以访问内存所有数据和所有外设，比如硬盘，网卡等等；

## 用户空间相关

- 用户空间：存放的是用户程序的代码和数据，处于虚拟空间；
- 用户态：当进程在执行用户自己的代码（非系统调用之类的函数）时，则称其处于用户态，CPU在特权级最低的3级用户代码中运行，当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态，因为中断处理程序将使用当前进程的内核栈；
- CPU堆栈指针寄存器指向：用户堆栈地址；
- 用户堆栈：进程处于用户态时使用的堆栈，存在于用户空间；
- 处于用户态进程的权利：处于用户态的进程，当它占有CPU的时候，只可以访问有限的内存，而且不允许访问外设，这里说的有限的内存其实就是用户空间，使用的是用户堆栈；

## 内核态和用户态的切换

- 系统调用

所有用户程序都是运行在用户态的，但是有时候程序确实需要做一些内核态的事情，例如从硬盘读取数据等。而唯一可以做这些事情的就是操作系统，所以此时程序就需要先操作系统的名义来执行这些操作。这时需要一个这样的机制：用户态程序切换到内核态，但是不能控制在内核态中执行的指令。这种机制叫系统调用，在CPU中的实现称之为陷阱指令(Trap Instruction)。

- 异常事件

当CPU正在执行运行在用户态的程序时，突然发生某些预先不可知的异常事件，这个时候就会触发从当前用户态执行的进程转向内核态执行相关的异常事件，典型的如缺页异常。

- 外围设备的中断

当外围设备完成用户的请求操作后，会像CPU发出中断信号，此时，CPU就会暂停执行下一条即将要执行的指令，转而去执行中断信号对应的处理程序，如果先前执行的指令是在用户态下，则自然就发生从用户态到内核态的转换。

注意：系统调用的本质其实也是中断，相对于外围设备的硬中断，这种中断称为软中断，这是操作系统为用户特别开放的一种中断，如Linux int 80h中断。所以从触发方式和效果上来看，这三种切换方式是完全一样的，都相当于是执行了一个中断响应的过程。但是从触发的对象来看，系统调用是进程主动请求切换的，而异常和硬中断则是被动的。

类型	源头	响应方式	处理机制
中断	外设	异步	持续，对用户应用程序是透明的
异常	应用程序意想不到的行为	同步	杀死或重新执行意想不到的应用程序指令
系统调用	应用程序请求操作提供服务	异步或同步	等待和持续

## Linux IO模式及 select、poll、epoll

### I/O 模型

一个输入操作通常包括两个阶段：

- 等待数据准备好
- 从内核向进程复制数据

对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核中的某个缓冲区。第二步就是把数据从内核缓冲区复制到应用进程缓冲区。

Unix 有五种 I/O 模型：

- 阻塞式 I/O
- 非阻塞式 I/O
- I/O 复用 (select 和 poll)
- 信号驱动式 I/O (SIGIO)
- 异步 I/O (AIO)

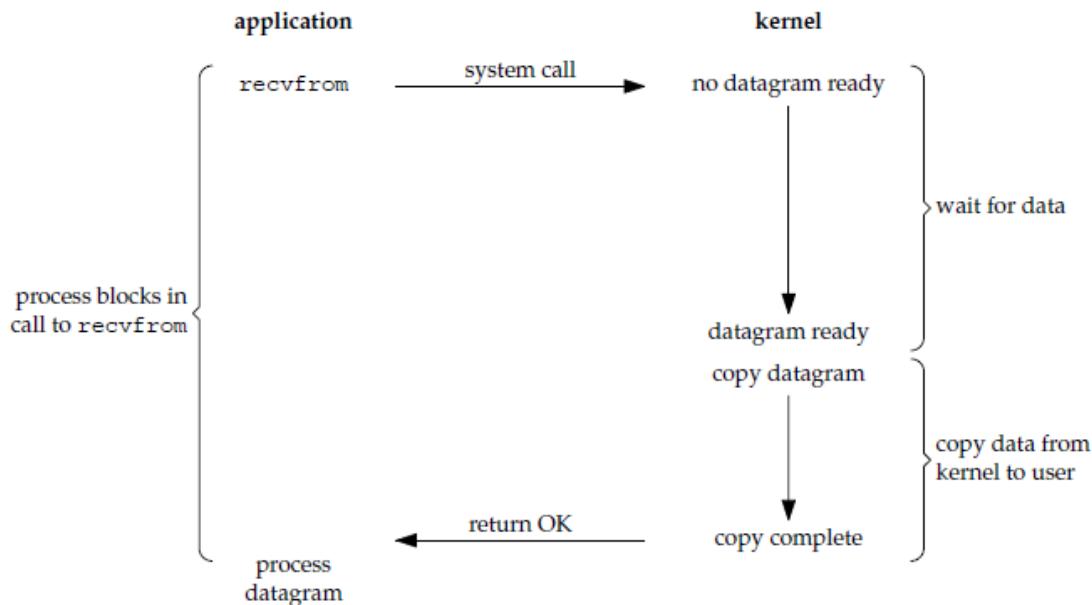
## 阻塞式 I/O

应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

应该注意到，在阻塞的过程中，其它应用进程还可以执行，因此阻塞不意味着整个操作系统都被阻塞。因为其它应用进程还可以执行，所以不消耗 CPU 时间，这种模型的 CPU 利用率会比较高。

下图中，recvfrom() 用于接收 Socket 传来的数据，并复制到应用进程的缓冲区 buf 中。这里把 recvfrom() 当成系统调用。

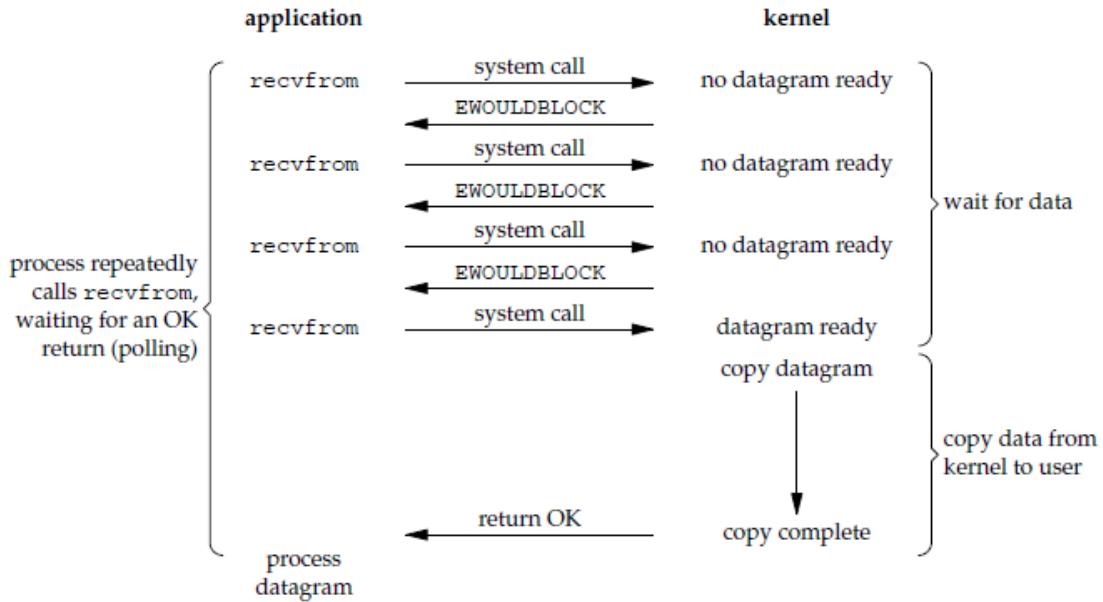
```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
```



## 非阻塞式 I/O

应用进程执行系统调用之后，内核返回一个错误码。应用进程可以继续执行，但是需要不断的执行系统调用来获知 I/O 是否完成，这种方式称为轮询（polling）。

由于 CPU 要处理更多的系统调用，因此这种模型的 CPU 利用率比较低。

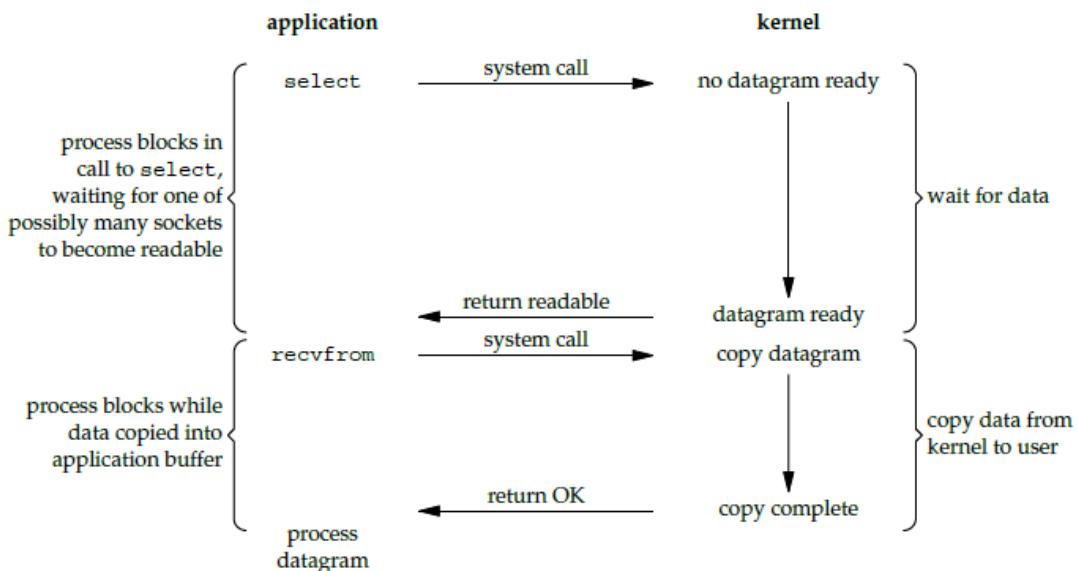


## I/O 复用

使用 `select` 或者 `poll` 等待数据，并且可以等待多个套接字中的任何一个变为可读。这一过程会被阻塞，当某一个套接字可读时返回，之后再使用 `recvfrom` 把数据从内核复制到进程中。

它可以让单个进程具有处理多个 I/O 事件的能力。又被称为 Event Driven I/O，即事件驱动 I/O。

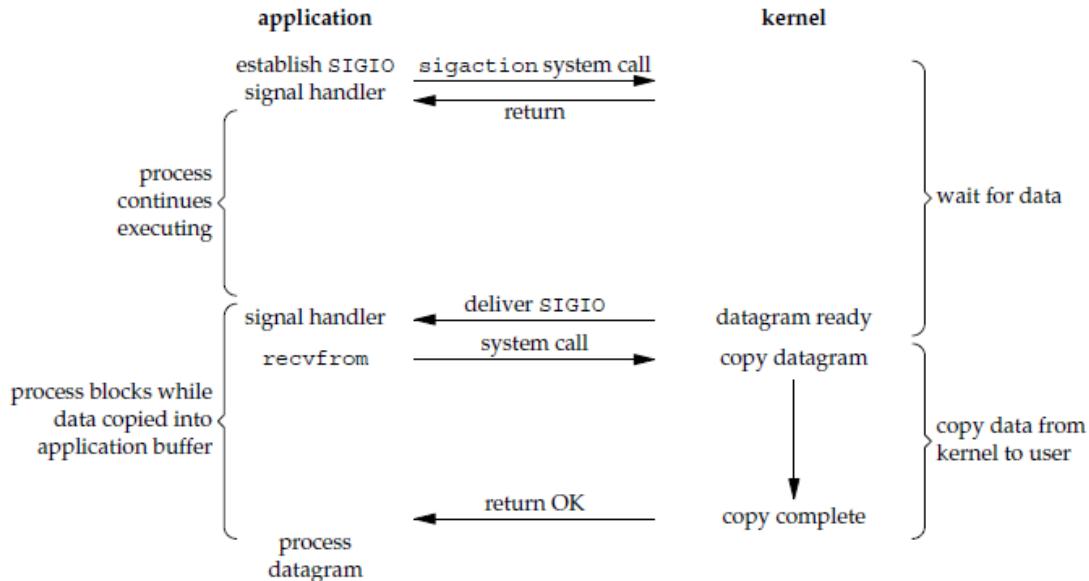
如果一个 Web 服务器没有 I/O 复用，那么每一个 Socket 连接都需要创建一个线程去处理。如果同时有几万个连接，那么就需要创建相同数量的线程。相比于多进程和多线程技术，I/O 复用不需要进程线程创建和切换的开销，系统开销更小。



## 信号驱动 I/O

应用进程使用 `sigaction` 系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送 `SIGIO` 信号，应用进程收到之后在信号处理程序中调用 `recvfrom` 将数据从内核复制到应用进程中。

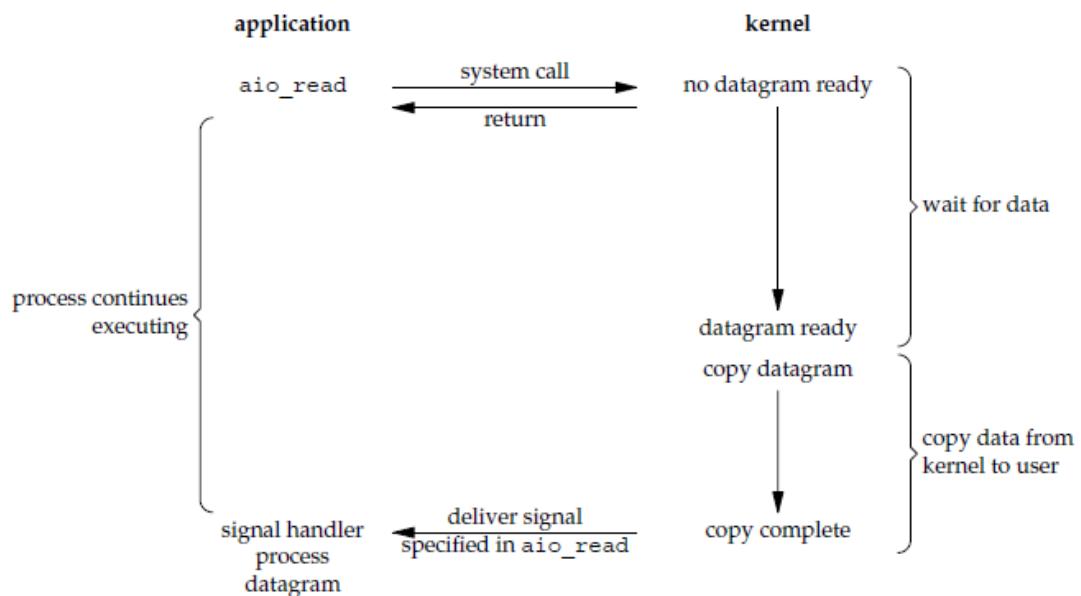
相比于非阻塞式 I/O 的轮询方式，信号驱动 I/O 的 CPU 利用率更高。



## 异步 I/O

应用进程执行 `aio_read` 系统调用会立即返回，应用进程可以继续执行，不会被阻塞，内核会在所有操作完成之后向应用进程发送信号。

异步 I/O 与信号驱动 I/O 的区别在于，异步 I/O 的信号是通知应用进程 I/O 完成，而信号驱动 I/O 的信号是通知应用进程可以开始 I/O。

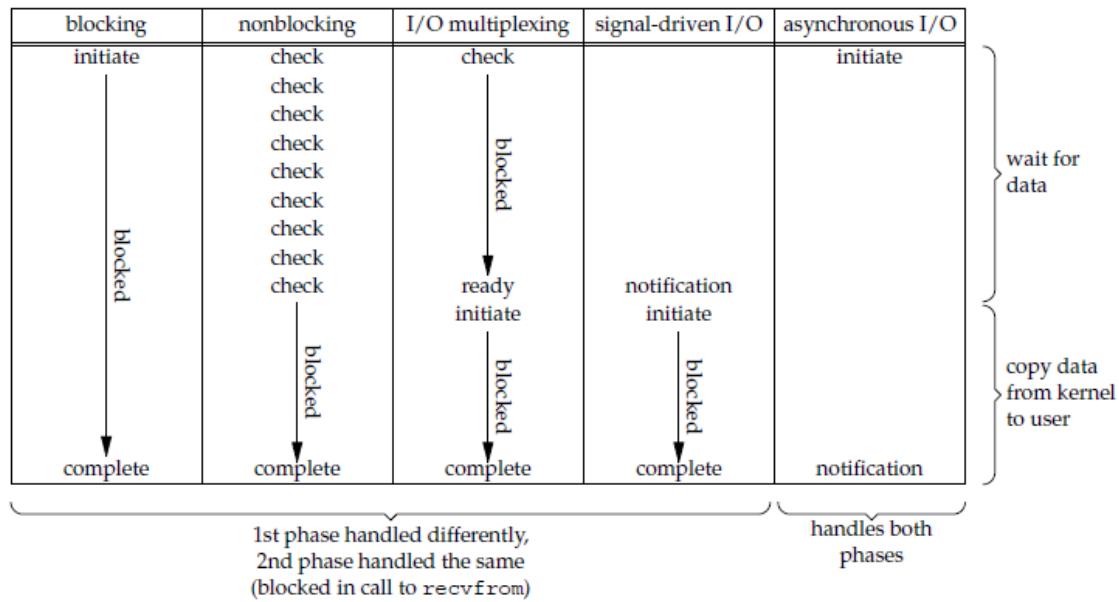


## 五大 I/O 模型比较

- 同步 I/O：将数据从内核缓冲区复制到应用进程缓冲区的阶段，应用进程会阻塞。
- 异步 I/O：不会阻塞。

阻塞式 I/O、非阻塞式 I/O、I/O 复用和信号驱动 I/O 都是同步 I/O，它们的主要区别在第一个阶段。

非阻塞式 I/O、信号驱动 I/O 和异步 I/O 在第一阶段不会阻塞。



## I/O 复用

select/poll/epoll 都是 I/O 多路复用的具体实现，select 出现的最早，之后是 poll，再是 epoll。

### select

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

有三种类型的描述符类型：readset、writeset、exceptset，分别对应读、写、异常条件的描述符集合。fd\_set 使用数组实现，数组大小使用 FD\_SETSIZE 定义。

timeout 为超时参数，调用 select 会一直阻塞直到有描述符的事件到达或者等待的时间超过 timeout。

成功调用返回结果大于 0，出错返回结果为 -1，超时返回结果为 0。

它仅仅知道有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

### poll

```
int poll(struct pollfd *fds, unsigned int nfds, int timeout);
```

poll本质上和select没有区别，它将用户传入的数据拷贝到内核空间，然后查询每个fd对应的设备状态，但是它没有最大连接数的限制，原因是它是基于链表来存储的。因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。

## 比较

### 1. 功能

select 和 poll 的功能基本相同，不过在一些实现细节上有所不同。

- select 会修改描述符，而 poll 不会；
- select 的描述符类型使用数组实现，FD\_SETSIZE 大小默认为 1024，因此默认只能监听 1024 个描述符。如果要监听更多描述符的话，需要修改 FD\_SETSIZE 之后重新编译；而 poll 的描述符类型使用链表实现，没有描述符数量的限制；

- poll 提供了更多的事件类型，并且对描述符的重复利用上比 select 高。
- 如果一个线程对某个描述符调用了 select 或者 poll，另一个线程关闭了该描述符，会导致调用结果不确定。

## 2. 速度

select 和 poll 速度都比较慢。

- select 和 poll 每次调用都需要将全部描述符从应用进程缓冲区复制到内核缓冲区。
- select 和 poll 的返回结果中没有声明哪些描述符已经准备好，所以如果返回值大于 0 时，应用进程都需要使用轮询的方式来找到 I/O 完成的描述符。

## 3. 可移植性

几乎所有的系统都支持 select，但是只有比较新的系统支持 poll。

### epoll

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

epoll\_ctl() 用于向内核注册新的描述符或者是改变某个文件描述符的状态。已注册的描述符在内核中会被维护在一棵红黑树上，通过回调函数内核会将 I/O 准备好的描述符加入到一个链表中管理，进程调用 epoll\_wait() 便可以得到事件完成的描述符。

从上面的描述可以看出，epoll 只需要将描述符从进程缓冲区向内核缓冲区拷贝一次，并且进程不需要通过轮询来获得事件完成的描述符。

epoll 仅适用于 Linux OS。

epoll 比 select 和 poll 更加灵活而且没有描述符数量限制。

epoll 对多线程编程更有友好，一个线程调用了 epoll\_wait() 另一个线程关闭了同一个描述符也不会产生像 select 和 poll 的不确定情况。

epoll 可以理解为 event poll，不同于忙轮询和无差别轮询，epoll 会把哪个流发生了怎样的 I/O 事件通知我们。所以我们说 epoll 实际上是事件驱动（每个事件关联上 fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了 O(1)）

## 工作模式

epoll 的描述符事件有两种触发模式：LT (level trigger) 和 ET (edge trigger)。

### 1. LT 模式

当 epoll\_wait() 检测到描述符事件到达时，将此事件通知进程，进程可以不立即处理该事件，下次调用 epoll\_wait() 会再次通知进程。是默认的一种模式，并且同时支持 Blocking 和 No-Blocking。

### 2. ET 模式

和 LT 模式不同的是，通知之后进程必须立即处理事件，下次再调用 epoll\_wait() 时不会再得到事件到达的通知。

很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。只支持 No-Blocking，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

## 应用场景

很容易产生一种错觉认为只要用 epoll 就可以了，select 和 poll 都已经过时了，其实它们都有各自的使用场景。

## 1. select 应用场景

select 的 timeout 参数精度为 1ns，而 poll 和 epoll 为 1ms，因此 select 更加适用于实时性要求比较高的场景，比如核反应堆的控制。

select 可移植性更好，几乎被所有主流平台所支持。

## 2. poll 应用场景

poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。

## 3. epoll 应用场景

只需要运行在 Linux 平台上，有大量的描述符需要同时轮询，并且这些连接最好是长连接。

需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。

需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll\_ctl() 进行系统调用，频繁系统调用降低效率。并且 epoll 的描述符存储在内核，不容易调试。

在连接数少并且连接都十分活跃的情况下，select 和 poll 的性能可能比 epoll 好，毕竟 epoll 的通知机制需要很多函数回调。

## 两句话总结

- select 和 poll 都需要轮询每个文件描述符，epoll 基于事件驱动，不用轮询
- select 和 poll 每次都需要拷贝文件描述符，epoll 不用
- select 最大连接数受限，epoll 和 poll 最大连接数不受限

tips: epoll 在内核中的实现，用红黑树管理事件块

## 命令

---

### 检索文件内容

```
grep "imooc" target*
```

查找以 target 开头的文件中找到 imooc 字符串

# 计算机操作系统

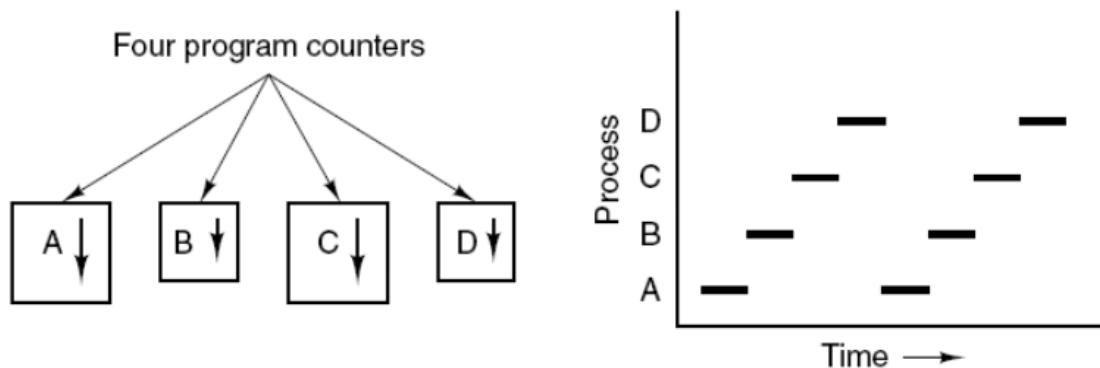
## 进程与线程

### 进程

进程是资源分配的基本单位。

进程控制块 (Process Control Block, PCB) 描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对 PCB 的操作。

下图显示了 4 个程序创建了 4 个进程，这 4 个进程可以并发地执行。

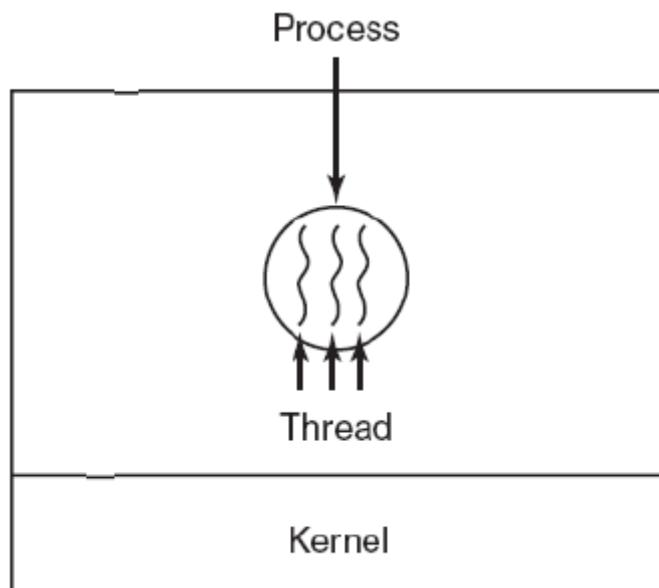


### 线程

线程是独立调度的基本单位。

一个进程中可以有多个线程，它们共享进程资源。

QQ 和浏览器是两个进程，浏览器进程里面有很多线程，例如 HTTP 请求线程、事件响应线程、渲染线程等等，线程的并发执行使得在浏览器中点击一个新链接从而发起 HTTP 请求时，浏览器还可以响应用户的其它事件。



### 区别

## I 拥有资源

进程是资源分配的基本单位，但是线程不拥有资源，线程可以访问隶属进程的资源。

## II 调度

线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。

## III 系统开销

由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。线程没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序比多线程的程序健壮。

## IV 通信方面

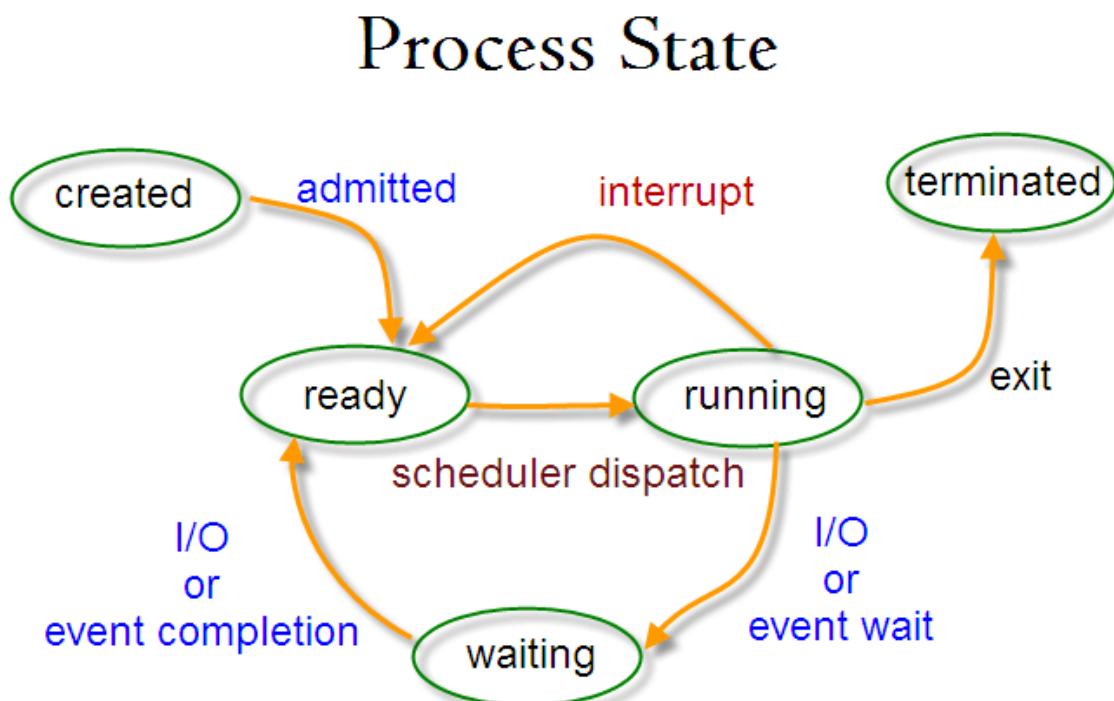
线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。

## 协程

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。协程的开销远远小于线程的开销。

## 进程状态切换



]

- 就绪状态 (ready)：等待被调度
- 运行状态 (running)
- 阻塞状态 (waiting)：等待资源

应该注意以下内容：

- 只有就绪态和运行态可以相互转换，其它的都是单向转换。就绪状态的进程通过调度算法从而获得 CPU 时间，转为运行状态；而运行状态的进程，在分配给它的 CPU 时间片用完之后就会转为就绪状态，等待下一次调度。
- 阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括 CPU 时间，缺少 CPU 时间会从运行态转换为就绪态。

## 进程调度算法

不同环境的调度算法目标不同，因此需要针对不同环境来讨论调度算法。

### 批处理系统

批处理系统没有太多的用户操作，在该系统中，调度算法目标是**保证吞吐量和周转时间**（从提交到终止的时间）。

#### 先来先服务 first-come first-served (FCFS)

非抢占式的调度算法，按照请求的顺序进行调度。

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。

#### 短作业优先 shortest job first (SJF)

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。

长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

#### 最短剩余时间优先 shortest remaining time next (SRTN)

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

### 交互式系统

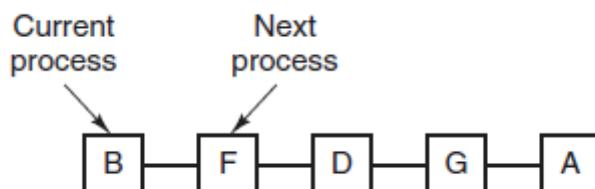
交互式系统有大量的用户交互操作，在该系统中调度算法的目标是**快速地进行响应**。

#### 时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。
- 而如果时间片过长，那么实时性就不能得到保证。



#### 优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

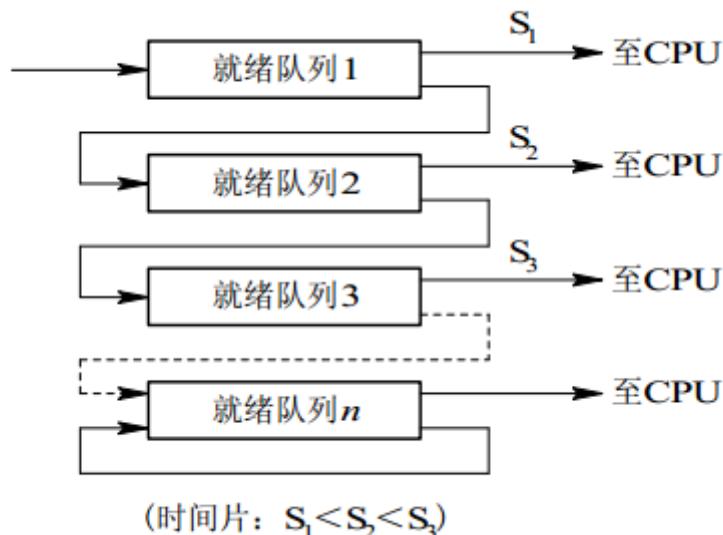
### 多级反馈队列

一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如 1,2,4,8,...。进程在第一个队列没执行完，就会被移到下一个队列。这种方式下，之前的进程只需要交换 7 次。

每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



## 实时系统

实时系统要求一个请求在一个确定时间内得到响应。

分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

## 进程同步

### 1. 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

### 2. 同步与互斥

- 同步：多个进程按一定顺序执行；
- 互斥：多个进程在同一时刻只有一个进程能进入临界区。

### 3. 信号量

信号量 (Semaphore) 是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

- **down**：如果信号量大于 0，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；

- **up**：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 **互斥量 (Mutex)**，0 表示临界区已经加锁，1 表示临界区解锁。

### 使用信号量实现生产者-消费者问题

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

## 4. 管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其它进程永远不能使用管程。

管程引入了 **条件变量** 以及相关的操作：**wait()** 和 **signal()** 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

## 进程通信

进程同步与进程通信很容易混淆，它们的区别在于：

- 进程同步：控制多个进程按一定顺序执行；
- 进程通信：进程间传输信息。

进程通信是一种手段，而进程同步是一种目的。也可以说，为了能够达到进程同步的目的，需要让进程进行通信，传输一些进程同步所需要的信息。

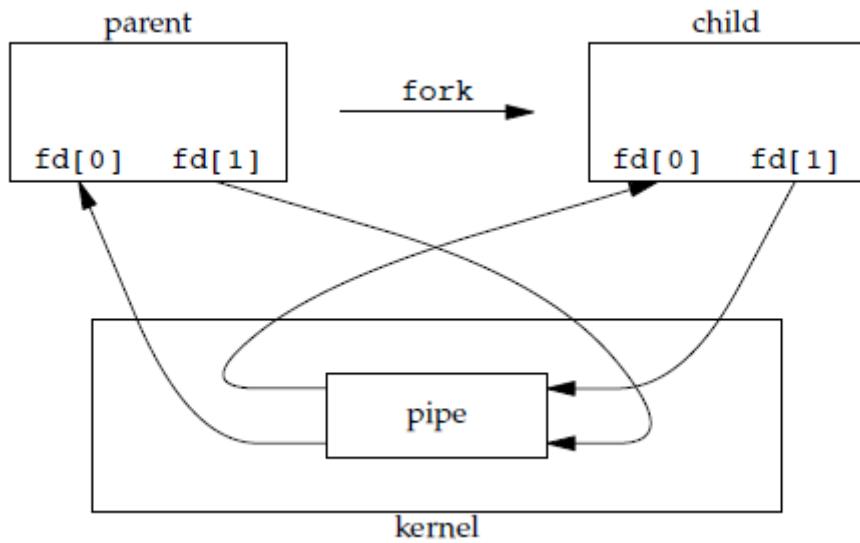
## 1. 管道

管道是通过调用 pipe 函数创建的，fd[0] 用于读，fd[1] 用于写。

```
#include <unistd.h>
int pipe(int fd[2]);
```

它具有以下限制：

- 只支持半双工通信（单向交替传输）；
- 只能在父子进程中使用。

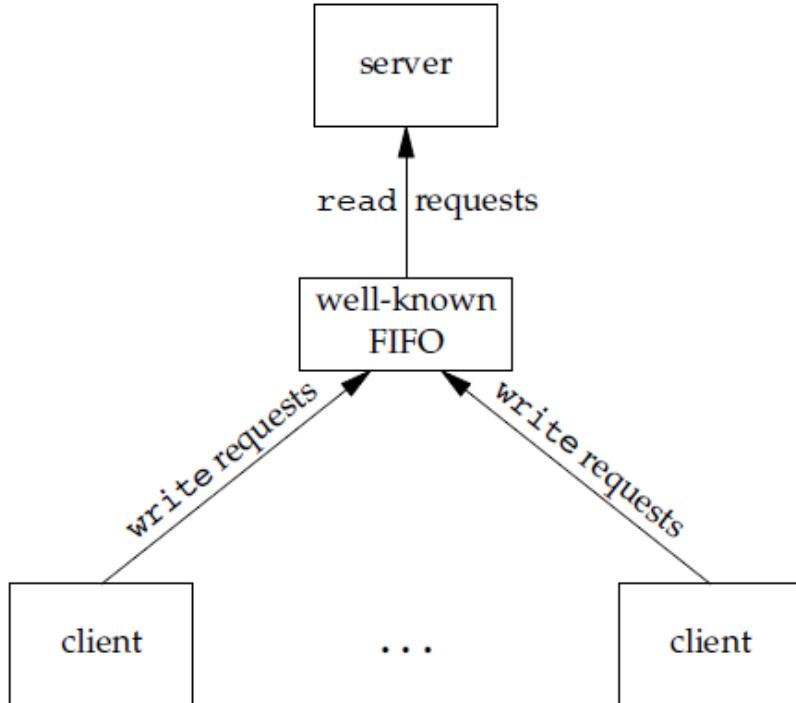


## 2. FIFO

也称为命名管道，去除了管道只能在父子进程中使用的限制。

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

FIFO 常用于客户-服务器应用程序中，FIFO 用作汇聚点，在客户进程和服务器进程之间传递数据。



## 3. 消息队列

相比于 FIFO，消息队列具有以下优点：

- 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；
- 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；

- 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。

## 4. 信号量

它是一个计数器，用于为多个进程提供对共享数据对象的访问。

## 5. 共享存储

允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。

需要使用信号量用来同步对共享存储的访问。

多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。

## 6. 套接字

与其它通信机制不同的是，它可用于不同机器间的进程通信。

# 死锁

---

## 必要条件

- 互斥：资源不能被共享，只能由一个进程使用。
- 请求与保持条件：进程已获得了一些资源，但因请求其它资源被阻塞时，对已获得的资源保持不放。
- 不可抢占：已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。
- 环路等待：有两个或者两个以上的进程组成一条环路，该环路中的每个进程都在等待下一个进程所占有的资源。

## 处理方法

主要有以下四种方法：

- 鸵鸟策略
- 死锁检测与死锁恢复
- 死锁预防
- 死锁避免

### 鸵鸟策略

把头埋在沙子里，假装根本没发生问题。

因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。

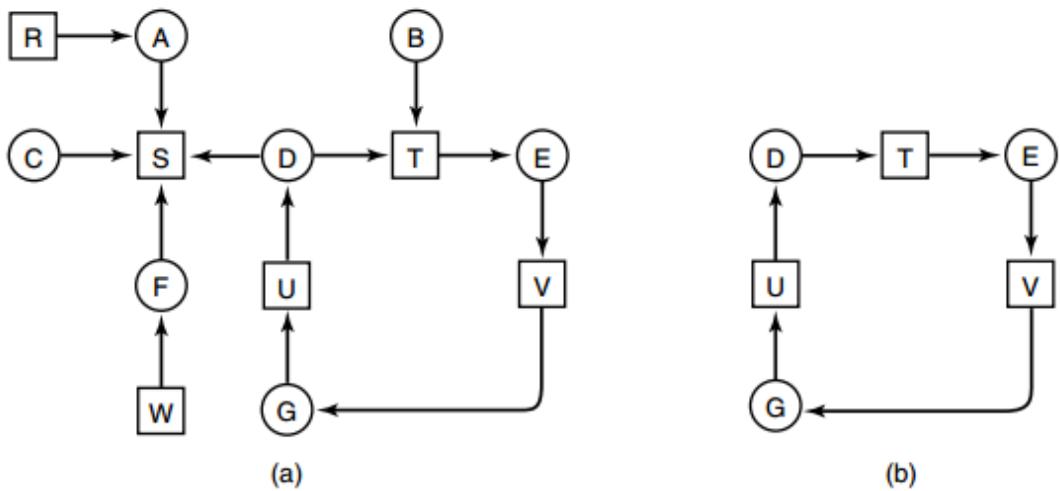
当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

大多数操作系统，包括 Unix, Linux 和 Windows，处理死锁问题的办法仅仅是忽略它。

### 死锁检测与死锁恢复

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

#### 1. 每种类型一个资源的死锁检测



**Figure 6-5.** (a) A resource graph. (b) A cycle extracted from (a).

上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先搜索，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死锁的发生。

## 2. 每种类型多个资源的死锁检测

	Tape drives	Plotters	Scanners	Blu-rays
E = (	4	2	3	1
A = (	2	1	0	0
Current allocation matrix				
C =	[ 0 0 1 0 ]	[ 2 0 0 1 ]	[ 0 1 2 0 ]	
Request matrix				
R =	[ 2 0 0 1 ]	[ 1 0 1 0 ]	[ 2 1 0 0 ]	

上图中，有三个进程四个资源，每个数据代表的含义如下：

- E 向量: 资源总量
  - A 向量: 资源剩余量
  - C 矩阵: 每个进程所拥有的资源数量, 每一行都代表一个进程拥有资源的数量
  - R 矩阵: 每个进程请求的资源数量

进程 P1 和 P2 所请求的资源都得不到满足，只有进程 P3 可以，让 P3 执行，之后释放 P3 拥有的资源，此时  $A = (2 \ 2 \ 2 \ 0)$ 。P2 可以执行，执行后释放 P2 拥有的资源， $A = (4 \ 2 \ 2 \ 1)$ 。P1 也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程  $P_i$ ，它所请求的资源小于等于  $A$ 。
2. 如果找到了这样一个进程，那么将  $C$  矩阵的第  $i$  行向量加到  $A$  中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

### 3. 死锁恢复

- 利用抢占恢复
- 利用回滚恢复
- 通过杀死进程恢复

## 死锁预防

在程序运行之前预防发生死锁。

### 1. 破坏互斥条件

例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。

### 2. 破坏占有和等待条件

一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。

### 3. 破坏不可抢占条件

### 4. 破坏环路等待

给资源统一编号，进程只能按编号顺序来请求资源。

## 死锁避免

在程序运行时避免发生死锁。

### 1. 安全状态

	Has	Max		Has	Max		Has	Max		Has	Max		Has	Max	
A	3	9		A	3	9		A	3	9		A	3	9	
B	2	4		B	4	4		B	0	-		B	0	-	
C	2	7		C	2	7		C	2	7		C	7	7	
	Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		
	(a)			(b)			(c)			(d)			(e)		

Figure 6-9. Demonstration that the state in (a) is safe.

图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

### 2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

**Figure 6-11.** Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

### 3. 多个资源的银行家算法

	Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still assigned

$E = (6342)$   
 $P = (5322)$   
 $A = (1020)$

**Figure 6-12.** The banker's algorithm with multiple resources.

上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态时安全的。

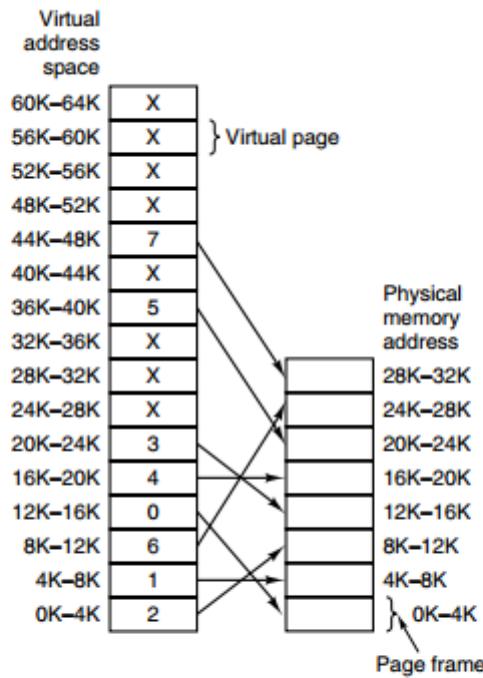
如果一个状态不是安全的，需要拒绝进入这个状态。

## 虚拟内存

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。

从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。



**Figure 3-9.** The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K to 12K means 8192-12287.

## 页面置换算法

在程序运行过程中，如果要访问的页面不在内存中，就发生缺页中断从而将该页调入内存中。此时如果内存已无空闲空间，系统必须从内存中调出一个页面到磁盘对换区中来腾出空间。

页面置换算法和缓存淘汰策略类似，可以将内存看成磁盘的缓存。在缓存系统中，缓存的大小有限，当有新的缓存到达时，需要淘汰一部分已经存在的缓存，这样才有空间存放新的缓存数据。

页面置换算法的主要目标是使页面置换频率最低（也可以说缺页率最低）。

### 1. 最佳

OPT, Optimal replacement algorithm

所选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率。

是一种理论上的算法，因为无法知道一个页面多长时间不再被访问。

举例：一个系统为某进程分配了三个物理块，并有如下页面引用序列：

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

开始运行时，先将 7, 0, 1 三个页面装入内存。当进程要访问页面 2 时，产生缺页中断，会将页面 7 换出，因为页面 7 再次被访问的时间最长。

### 2. 最近最久未使用

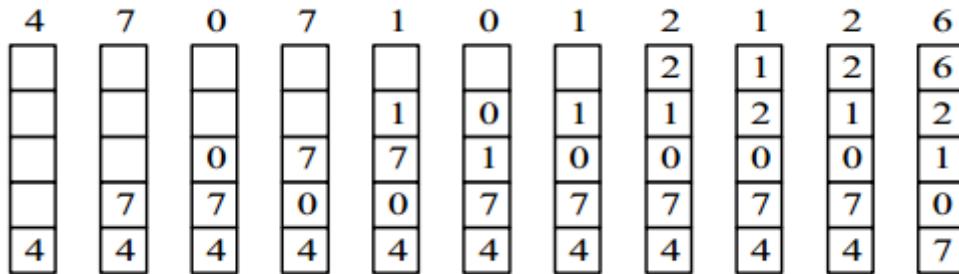
## LRU, Least Recently Used

虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面的情况。LRU 将最近最久未使用的页面换出。

为了实现 LRU，需要在内存中维护一个所有页面的链表。当一个页面被访问时，将这个页面移到链表表头。这样就能保证链表表尾的页面是最近最久未访问的。

因为每次访问都需要更新链表，因此这种方式实现的 LRU 代价很高。

4, 7, 0, 7, 1, 0, 1, 2, 1, 2, 6



## 3. 最近未使用

### NRU, Not Recently Used

每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：

- R=0, M=0
- R=0, M=1
- R=1, M=0
- R=1, M=1

当发生缺页中断时，NRU 算法随机地从类编号最小的非空类中挑选一个页面将它换出。

NRU 优先换出已经被修改的脏页面 (R=0, M=1)，而不是被频繁使用的干净页面 (R=1, M=0)。

## 4. 先进先出

### FIFO, First In First Out

选择换出的页面是最先进入的页面。

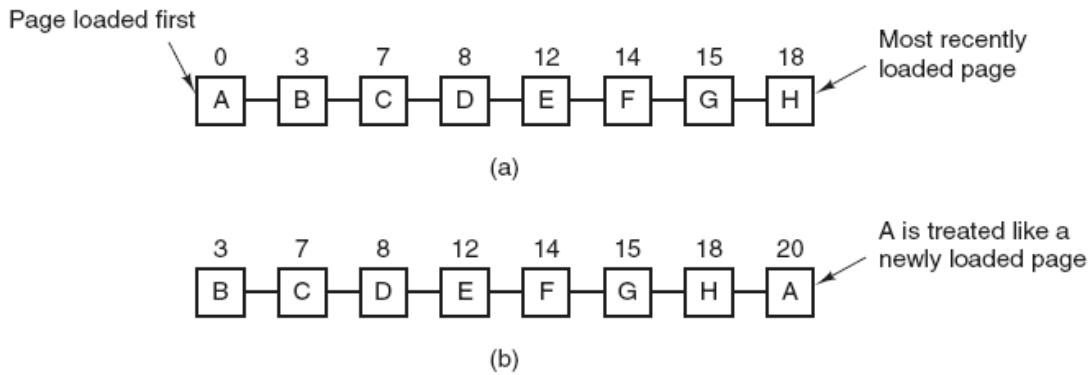
该算法会将那些经常被访问的页面也被换出，从而使缺页率升高。

## 5. 第二次机会算法

FIFO 算法可能会把经常使用的页面置换出去，为了避免这一问题，对该算法做一个简单的修改：

当页面被访问(读或写)时设置该页面的 R 位为 1。需要替换的时候，检查最老页面的 R 位。如果 R 位是 0，那么这个页面既老又没有被使用，可以立刻置换掉；如果是 1，就将 R 位清 0，并把该页面放到链表的尾端，修改它的装入时间使它就像刚装入的一样，然后继续从链表的头部开始搜索。

[

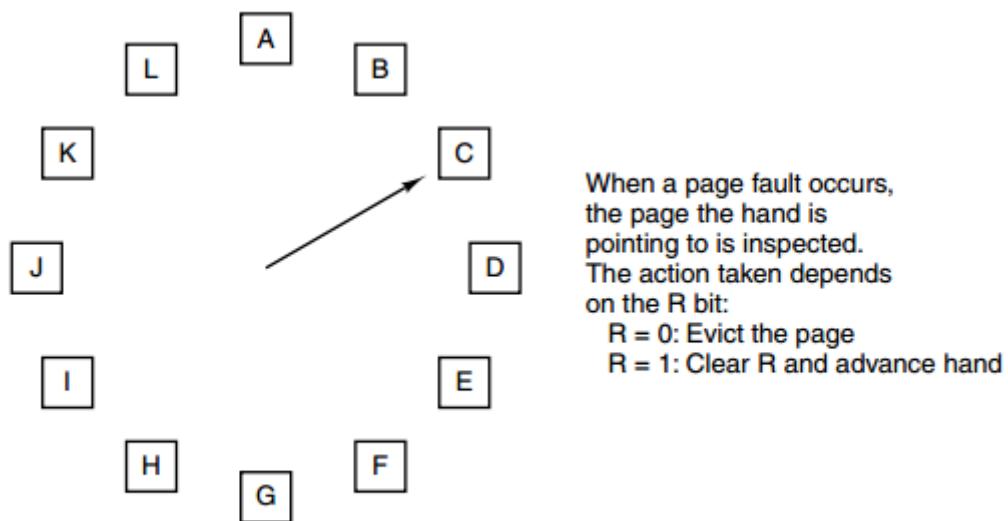


**Figure 3-15.** Operation of second chance. (a) Pages sorted in FIFO order.  
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

## 6. 时钟

### Clock

第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环形链表将页面连接起来，再使用一个指针指向最老的页面。



**Figure 3-16.** The clock page replacement algorithm.

## 段页式与分段、分页的比较

程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间划分成大小相同的页。这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能。

- 对程序员的透明性：分页透明，但是分段需要程序员显式划分每个段。
- 地址空间的维度：分页是一维地址空间，分段是二维的。
- 大小是否可以改变：页的大小不可变，段的大小可以动态改变。
- 出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

## 静态链接库和动态链接库

静态库：这类库的名字一般是libxxx.a，xxx为库的名字。利用静态函数库编译成的文件比较大，因为整个函数库的所有数据都会被整合进目标代码中，他的优点就显而易见了，即编译后的执行程序不需要外部的函数库支持，因为所有使用的函数都已经被编译进去了。当然这也会成为他的缺点，因为如果静态函数库改变了，那么你的程序必须重新编译。

动态库：这类库的名字一般是libxxx.M.N.so，同样的xxx为库的名字，M是库的主版本号，N是库的副版本号。当然也可以不要版本号，但名字必须有。相对于静态函数库，动态函数库在编译的时候并没有被编译进目标代码中，你的程序执行到相关函数时才调用该函数库里的相应函数，因此动态函数库所产生的可执行文件比较小。由于函数库没有被整合进你的程序，而是程序运行时动态的申请并调用，所以程序的运行环境中必须提供相应的库。动态函数库的改变并不影响你的程序，所以动态函数库的升级比较方便。linux系统有几个重要的目录存放相应的函数库，如/lib /usr/lib。

# 红黑树

---

- 红黑树特点:
  1. 每个节点非红即黑；
  2. 根节点总是黑色的；
  3. 每个叶子节点都是黑色的空节点（NIL节点）；
  4. 如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；
  5. 从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）。

- 红黑树的应用：

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。

- 为什么要用红黑树

简单来说红黑树就是为了解决二叉查找树和平衡二叉树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构，平衡二叉树为了维护其严格平衡性，旋转操作代价太大。详细了解可以查看[漫画：什么是红黑树？](#)（也介绍到了二叉查找树，非常推荐）

# 排序

## 排序算法比较

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

- 快速排序递归

```
package sort.quicksort;

import java.util.Arrays;

/**
 * 快速排序递归
 */
public class QuickSortRecursion {

    private static void quicksort(int[] arr) {
        int n = arr.length;
        quicksort(arr, 0, n - 1);
    }

    private static void quicksort(int[] arr, int start, int end) {
        if (arr == null || start >= end) {
            return;
        }
        int i = start, j = end;
        int pivotKey = arr[start];
        while (i < j) {
            while (i < j && arr[j] >= pivotKey) {
                j--;
            }
            if (i < j) {
                arr[i++] = arr[j];
            }
        }
    }
}
```

```

        }
        while (i < j && arr[i] <= pivotKey) {
            i++;
        }
        if (i < j) {
            arr[j--] = arr[i];
        }
    }
    arr[i] = pivotKey;
    quicksort(arr, start, i - 1);
    quicksort(arr, i + 1, end);
}

public static void main(String[] args) {
    int[] arr = {10, 5, 2, 3};
    quicksort(arr);
    System.out.println(Arrays.toString(arr)); // [2, 3, 5, 10]
}
}

```

- 归并排序递归

```

package sort.mergesort;

import java.util.Arrays;

/**
 * 归并排序递归
 */
public class MergeSortRecursion {
    public static void sort(int[] arr) {
        int n = arr.length;
        sort(arr, 0, n - 1);
    }

    // 递归使用归并排序,对arr[l...r]的范围进行排序
    private static void sort(int[] arr, int l, int r) {
        if (l >= r)
            return;

        int mid = (l + r) / 2;
        sort(arr, l, mid);
        sort(arr, mid + 1, r);
        merge(arr, l, mid, r);
    }

    // 将arr[l...mid]和arr[mid+1...r]两部分进行归并
    private static void merge(int[] arr, int l, int mid, int r) {
        int[] aux = Arrays.copyOfRange(arr, l, r + 1);

        // 初始化, i指向左半部分的起始索引位置l; j指向右半部分起始索引位置mid+1
        int i = l, j = mid + 1;
        for (int k = l; k <= r; k++) {

            if (i > mid) { // 如果左半部分元素已经全部处理完毕
                arr[k] = aux[j - 1];
                j++;
            }
        }
    }
}

```

```

        } else if (j > r) { // 如果右半部分元素已经全部处理完毕
            arr[k] = aux[i - 1];
            i++;
        } else if (aux[i - 1] - aux[j - 1] < 0) { // 左半部分所指元素 < 右半部
分所指元素
            arr[k] = aux[i - 1];
            i++;
        } else { // 左半部分所指元素 >= 右半部分所指元素
            arr[k] = aux[j - 1];
            j++;
        }
    }
}

public static void main(String[] args) {
    int[] arr = {10, 5, 2, 3};
    sort(arr);
    System.out.println(Arrays.toString(arr)); // [2, 3, 5, 10]
}
}

```

- 堆排序

```

package sort.heapsort;

import java.util.Arrays;

public class HeapSort {

    public static void heapsort(int[] arr) {
        int n = arr.length;
        // 注意，此时我们的堆是从0开始索引的
        // 从(最后一个元素的索引-1)/2开始
        // 最后一个元素的索引 = n-1
        for (int i = (n - 1 - 1) / 2; i >= 0; i--)
            shiftDown(arr, n, i);

        for (int i = n - 1; i > 0; i--) {
            swap(arr, 0, i);
            shiftDown(arr, i, 0);
        }
    }

    // 交换堆中索引为i和j的两个元素
    private static void swap(int[] arr, int i, int j) {
        int t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
    }

    // 优化的shiftDown过程，使用赋值的方式取代不断的swap，
    // 该优化思想和我们之前对插入排序进行优化的思路是一致的
    private static void shiftDown(int[] arr, int n, int k) {

        int e = arr[k];
        while (2 * k + 1 < n) {
            int j = 2 * k + 1;
            if (j + 1 < n && arr[j + 1] > arr[j])
                j++;
            if (e < arr[j])
                break;
            arr[k] = arr[j];
            k = j;
        }
        arr[k] = e;
    }
}

```

```
    if (j + 1 < n && arr[j + 1] - arr[j] > 0)
        j += 1;

    if (e - arr[j] >= 0)
        break;

    arr[k] = arr[j];
    k = j;
}
arr[k] = e;
}

// 测试 Heapsort
public static void main(String[] args) {
    int[] arr = {10, 5, 2, 3};
    heapsort(arr);
    System.out.println(Arrays.toString(arr)); // [2, 3, 5, 10]
}
}
```

# 树前中后序遍历

```
package tree;

import java.util.Stack;

public class PreInPosTraversal {

    public static class Node {
        public int value;
        public Node left;
        public Node right;

        public Node(int data) {
            this.value = data;
        }
    }

    public static void preOrderRecur(Node head) {
        if (head == null) {
            return;
        }
        System.out.print(head.value + " ");
        preOrderRecur(head.left);
        preOrderRecur(head.right);
    }

    public static void inOrderRecur(Node head) {
        if (head == null) {
            return;
        }
        inOrderRecur(head.left);
        System.out.print(head.value + " ");
        inOrderRecur(head.right);
    }

    public static void posOrderRecur(Node head) {
        if (head == null) {
            return;
        }
        posOrderRecur(head.left);
        posOrderRecur(head.right);
        System.out.print(head.value + " ");
    }

    public static void preOrderUnRecur(Node head) {
        System.out.print("pre-order: ");
        if (head != null) {
            Stack<Node> stack = new Stack<Node>();
            stack.add(head);
            while (!stack.isEmpty()) {
                head = stack.pop();
                System.out.print(head.value + " ");
            }
        }
    }
}
```

```

        if (head.right != null) {
            stack.push(head.right);
        }
        if (head.left != null) {
            stack.push(head.left);
        }
    }
    System.out.println();
}

public static void inOrderUnRecur(Node head) {
    System.out.print("in-order: ");
    if (head != null) {
        Stack<Node> stack = new Stack<Node>();
        while (!stack.isEmpty() || head != null) {
            if (head != null) {
                stack.push(head);
                head = head.left;
            } else {
                head = stack.pop();
                System.out.print(head.value + " ");
                head = head.right;
            }
        }
    }
    System.out.println();
}

public static void posOrderUnRecur1(Node head) {
    System.out.print("pos-order: ");
    if (head != null) {
        Stack<Node> s1 = new Stack<Node>();
        Stack<Node> s2 = new Stack<Node>();
        s1.push(head);
        while (!s1.isEmpty()) {
            head = s1.pop();
            s2.push(head);
            if (head.left != null) {
                s1.push(head.left);
            }
            if (head.right != null) {
                s1.push(head.right);
            }
        }
        while (!s2.isEmpty()) {
            System.out.print(s2.pop().value + " ");
        }
    }
    System.out.println();
}

public static void posOrderUnRecur2(Node h) {
    System.out.print("pos-order: ");
    if (h != null) {
        Stack<Node> stack = new Stack<Node>();
        stack.push(h);
        Node c = null;
    }
}

```

```

        while (!stack.isEmpty()) {
            c = stack.peek();
            if (c.left != null && h != c.left && h != c.right) {
                stack.push(c.left);
            } else if (c.right != null && h != c.right) {
                stack.push(c.right);
            } else {
                System.out.print(stack.pop().value + " ");
                h = c;
            }
        }
    }
    System.out.println();
}

public static void main(String[] args) {
    Node head = new Node(5);
    head.left = new Node(3);
    head.right = new Node(8);
    head.left.left = new Node(2);
    head.left.right = new Node(4);
    head.left.left.left = new Node(1);
    head.right.left = new Node(7);
    head.right.left.left = new Node(6);
    head.right.right = new Node(10);
    head.right.right.left = new Node(9);
    head.right.right.right = new Node(11);

    // recursive
    System.out.println("=====recursive=====");
    System.out.print("pre-order: ");
    preOrderRecur(head);
    System.out.println();
    System.out.print("in-order: ");
    inOrderRecur(head);
    System.out.println();
    System.out.print("pos-order: ");
    posOrderRecur(head);
    System.out.println();

    // unrecrusive
    System.out.println("=====unrecrusive=====");
    preOrderUnRecur(head);
    inOrderUnRecur(head);
    posOrderUnRecur1(head);
    posOrderUnRecur2(head);
}

}

```

# MySQL

---

## REDO日志和UNDO日志

---

### Undo Log

Undo Log是为了实现事务的原子性，在MySQL数据库InnoDB存储引擎中，还用了Undo Log来实现多版本并发控制(简称：MVCC)。

事务的原子性(Atomicity)事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。

原理Undo Log的原理很简单，为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为Undo Log）。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。

之所以能同时保证原子性和持久化，是因为以下特点：

更新数据前记录。

Undo log为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。

Undo log必须先于数据持久化到磁盘。如果在G,H之间系统崩溃，undo log是完整的，可以用来回滚事务。

如果在A-F之间系统崩溃，因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。

缺陷：每个事务提交前将数据和Undo Log写入磁盘，这样会导致大量的磁盘IO，因此性能很低。如果能够将数据缓存一段时间，就能减少IO提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即Redo Log。

### Redo Log:

原理和Undo Log相反，Redo Log记录的是新数据的备份。在事务提交前，只要将Redo Log持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到最新的状态。

## ACID

---

### 1. 原子性 (Atomicity)

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

### 2. 一致性 (Consistency)

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的。

### 3. 隔离性 (Isolation)

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

### 4. 持久性 (Durability)

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

使用重做日志来保证持久性。

## 封锁

### 封锁粒度

MySQL 中提供了两种封锁粒度：行级锁以及表级锁。

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁、以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统开销就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

### 封锁类型

#### 1. 读写锁

- 排它锁 (Exclusive)，简写为 X 锁，又称写锁。
- 共享锁 (Shared)，简写为 S 锁，又称读锁。

有以下两个规定：

- 一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。
- 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

锁的兼容关系如下：

-	X	S
X	✗	✗
S	✗	✓

#### 2. 意向锁

使用意向锁 (Intention Locks) 可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。

意向锁在原来的 X/S 锁之上引入了 IX/IS，IX/IS 都是表锁，用来表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。有以下两个规定：

- 一个事务在获得某个数据行对象的 S 锁之前，必须先获得表的 IS 锁或者更强的锁；

- 一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁。

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：

-	X	IX	S	IS
X	×	×	×	×
IX	×	√	×	√
S	×	×	√	√
IS	×	√	√	√

解释如下：

- 任意 IS/IX 锁之间都是兼容的，因为它们只是表示想要对表加锁，而不是真正加锁；
- S 锁只与 S 锁和 IS 锁兼容，也就是说事务 T 想要对数据行加 S 锁，其它事务可以已经获得对表或者表中的行的 S 锁。

## 封锁协议

### 1. 三级封锁协议

#### 一级封锁协议

事务 T 要修改数据 A 时必须加 X 锁，直到 T 结束才释放锁。

可以解决丢失修改问题，因为不能同时有两个事务对同一个数据进行修改，那么事务的修改就不会被覆盖。

T1	T2
lock-x(A)	
read A=20	
	lock-x(A)
	wait
write A=19	.
commit	.
unlock-x(A)	.
	obtain
	read A=19
	write A=21
	commit
	unlock-x(A)

## 二级封锁协议

在一级的基础上，要求读取数据 A 时必须加 S 锁，读取完马上释放 S 锁。

可以解决读脏数据问题，因为如果一个事务在对数据 A 进行修改，根据 1 级封锁协议，会加 X 锁，那么就不能再加 S 锁了，也就是不会读入数据。

T1	T2
lock-x(A)	
read A=20	
write A=19	
	lock-s(A)
	wait
rollback	.
A=20	.
unlock-x(A)	.
	obtain
	read A=20
	unlock-s(A)
	commit

## 三级封锁协议

在二级的基础上，要求读取数据 A 时必须加 S 锁，直到事务结束了才能释放 S 锁。

可以解决不可重复读的问题，因为读 A 时，其它事务不能对 A 加 X 锁，从而避免了在读的期间数据发生改变。

T1	T2
lock-s(A)	
read A=20	
	lock-x(A)
	wait
read A=20	.
commit	.
unlock-s(A)	.
	obtain
	read A=20
	write A=19
	commit
	unlock-X(A)

## 2.两段锁协议

加锁和解锁分为两个阶段进行。

可串行化调度是指，通过并发控制，使得并发执行的事务结果与某个串行执行的事务结果相同。

事务遵循两段锁协议是保证可串行化调度的充分条件。例如以下操作满足两段锁协议，它是可串行化调度。

```
lock-x(A) ... lock-s(B) ... lock-s(C) ... unlock(A) ... unlock(C) ... unlock(B)
```

但不是必要条件，例如以下操作不满足两段锁协议，但是它还是可串行化调度。

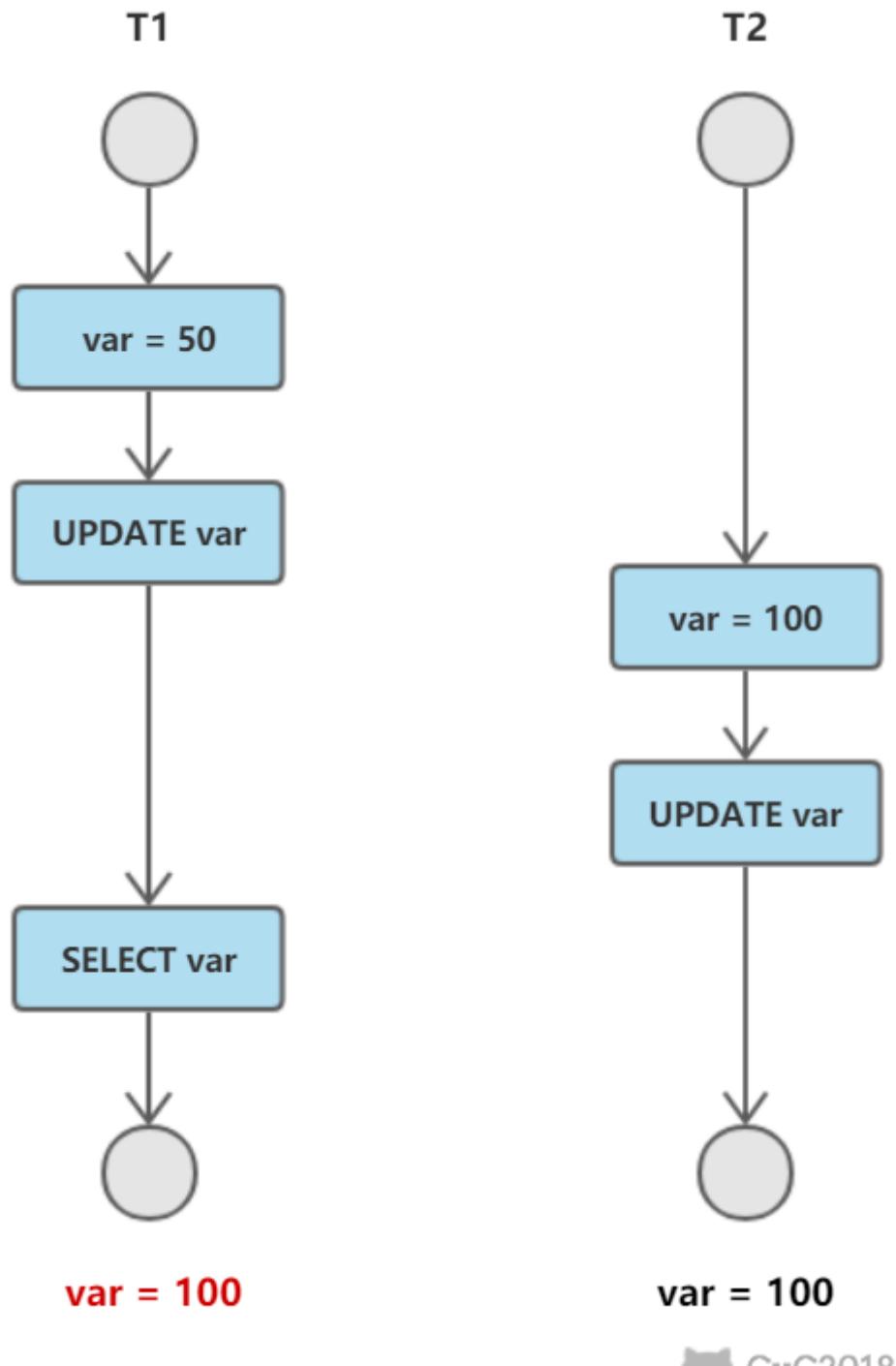
```
lock-x(A) ... unlock(A) ... lock-s(B) ... unlock(B) ... lock-s(C) ... unlock(C)
```

## 并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

### 丢失修改

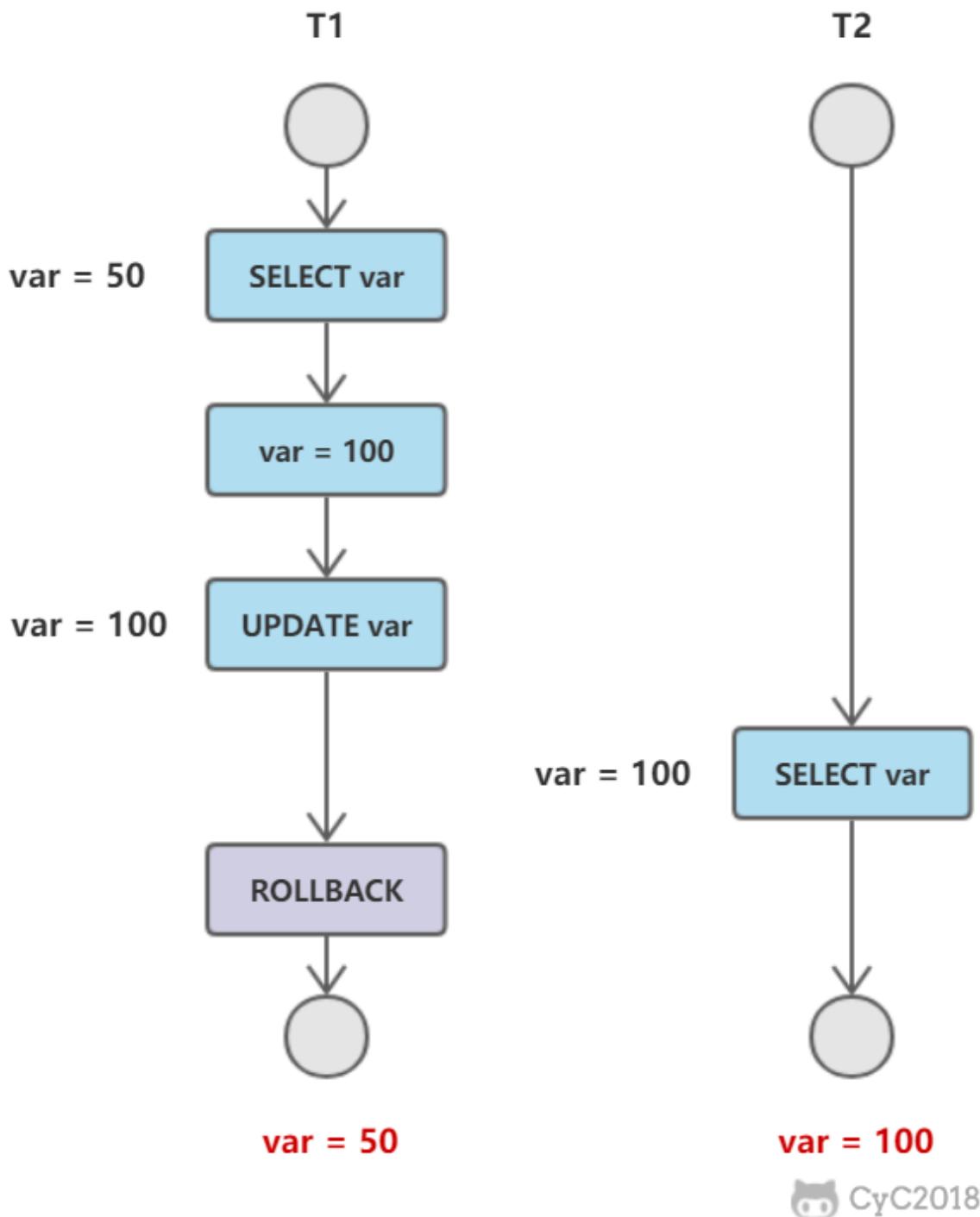
T1 和 T2 两个事务都对一个数据进行修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。



## 读脏数据

T1 修改一个数据，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。



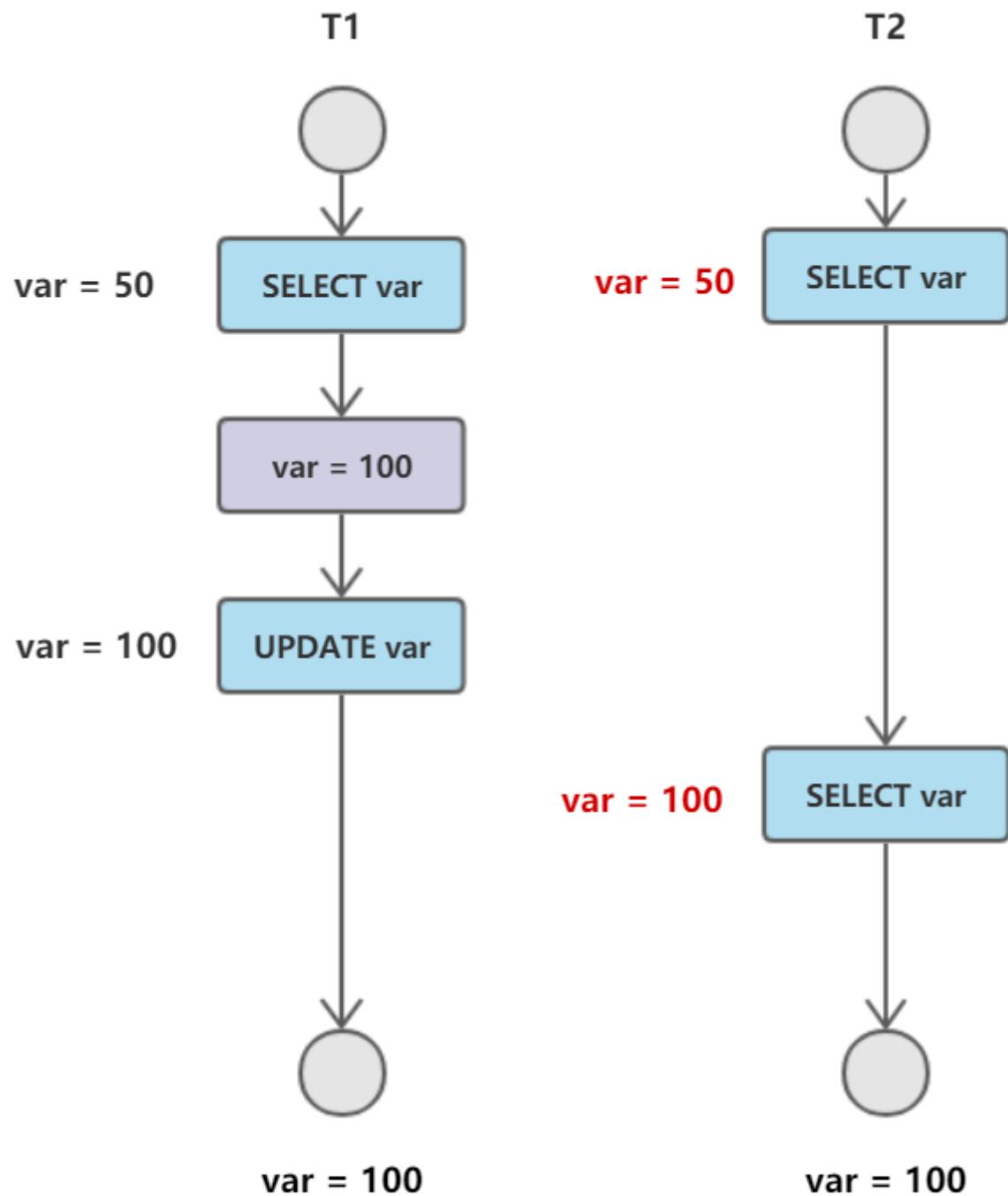


## 不可重复读

T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。



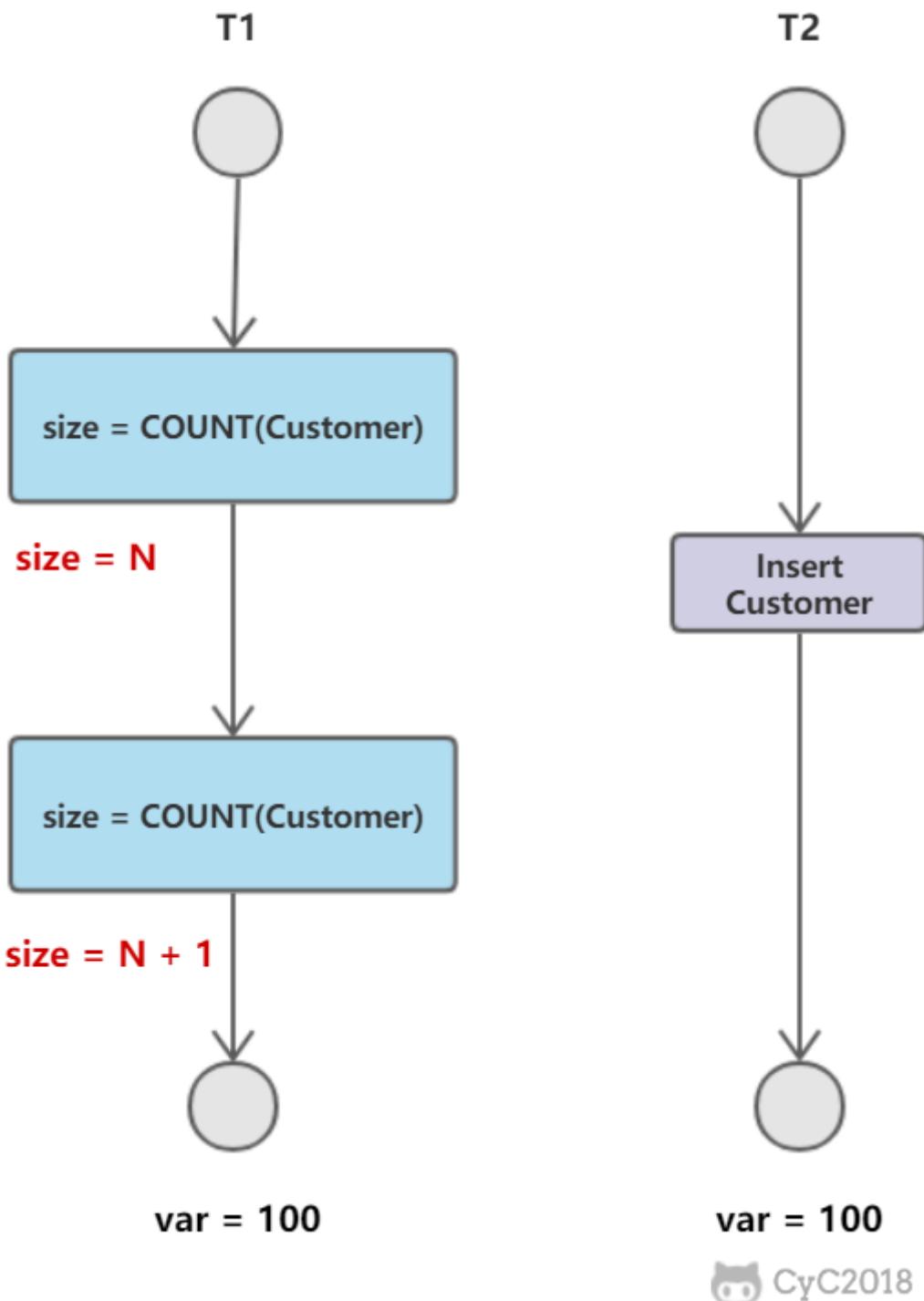
CyC2018



 CyC2018

## 幻影读

T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和第一次读取的结果不同。



产生并发不一致性问题主要原因是破坏了事务的隔离性，解决方法是通过并发控制来保证隔离性。并发控制可以通过封锁来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了事务的隔离级别，让用户以一种更轻松的方式处理并发一致性问题。

## 隔离级别

### 未提交读 (READ UNCOMMITTED)

事务中的修改，即使没有提交，对其他事务也是可见的。

### 提交读 (READ COMMITTED)

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其他事务是不可见的。

## 可重复读 (REPEATABLE READ)

保证在同一个事务中多次读取同样数据的结果是一样的。

## 可串行化 (SERIALIZABLE)

强制事务串行执行。

需要加锁实现，而其它隔离级别通常不需要。

隔离级别	脏读	不可重复读	幻影读
未提交读	√	√	√
提交读	✗	√	√
可重复读	✗	✗	√
可串行化	✗	✗	✗

## 多版本并发控制

多版本并发控制 (Multi-Version Concurrency Control, MVCC) 是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，**用于实现提交读和可重复读这两种隔离级别**。而未提交读隔离级别总是读取最新的数据行，无需使用 MVCC。可串行化隔离级别需要对所有读取的行都加锁，单纯使用 MVCC 无法实现。

### 准备

测试环境：Mysql 5.7.20-log

数据库默认隔离级别：**RR (Repeatable Read, 可重复读)**，MVCC主要适用于Mysql的RC,RR隔离级别

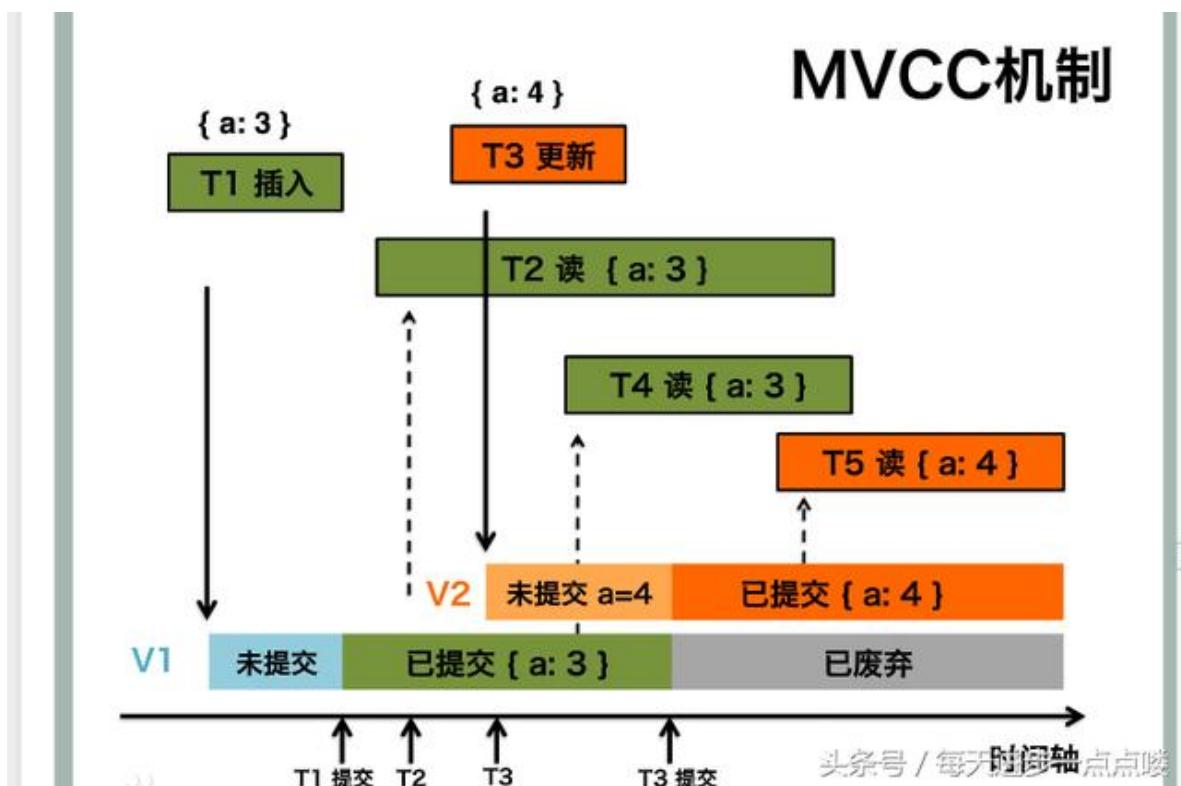
创建一张存储引擎为testmvcc的表，sql为：

```
CREATE TABLE testmvcc (
    id int(11) DEFAULT NULL,
    name varchar(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

### MVCC

英文全称为Multi-Version Concurrency Control,翻译为中文即 多版本并发控制。在小编看来，他无非就是乐观锁的一种实现方式。在Java编程中，如果把乐观锁看成一个接口，MVCC便是这个接口的一个实现类而已。

# MVCC机制



## 基本原理

MVCC的实现，通过保存数据在某个时间点的快照来实现的。这意味着一个事务无论运行多长时间，在同一个事务里能够看到数据一致的视图。根据事务开始的时间不同，同时也意味着在同一个时刻不同事务看到的相同表里的数据可能是不同的。

## 基本特征

- 每行数据都存在一个版本，每次数据更新时都更新该版本。
- 修改时Copy出当前版本随意修改，各个事务之间无干扰。
- 保存时比较版本号，如果成功 (commit)，则覆盖原记录；失败则放弃copy (rollback)

## InnoDB存储引擎MVCC的实现策略

在每一行数据中额外保存两个隐藏的列：当前行创建时的版本号和删除时的版本号（可能为空，其实还有一列称为回滚指针，用于事务回滚，不在本文范畴）。这里的版本号并不是实际的时间值，而是系统版本号。每开始新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询每行记录的版本号进行比较。

每个事务又有自己的版本号，这样事务内执行CRUD操作时，就通过版本号的比较来达到数据版本控制的目的。

## MVCC下InnoDB的增删查改是怎么work的

1.插入数据 (insert) :记录的版本号即当前事务的版本号，执行一条数据语句：`insert into testmvcc values(1,"test");`

假设事务id为1，那么插入后的数据行如下：

id	name	create version	delete version
1	test	1	

2、在更新操作的时候，采用的是先标记旧的那行记录为已删除，并且删除版本号是事务版本号，然后插入一行新的记录的方式。

比如，针对上面那行记录，事务Id为2 要把name字段更新update table set name= 'new\_value' where id=1;

id	name	create version	delete version
1	test	1	2
1	new_value	2	

头条号 / 每天进步一点点

3、删除操作的时候，就把事务版本号作为删除版本号。比如delete from table where id=1;

id	name	create version	delete version
1	new_value	2	3

头条号 / 每天进步一点点

4、查询操作：

从上面的描述可以看到，在查询时要符合以下两个条件的记录才能被事务查询出来：

1) 删除版本号未指定或者大于当前事务版本号，即查询事务开启后确保读取的行未被删除。(即上述事务id为2的事务查询时，依然能读取到事务id为3所删除的数据行)

2) 创建版本号 小于或者等于 当前事务版本号，就是说记录创建是在当前事务中（等于的情况）或者在当前事务启动之前的其他事物进行的Insert。

(即事务id为2的事务只能读取到create version<=2的已提交的事务的数据集)

补充：

1.MVCC手段只适用于MySQL隔离级别中的读已提交（Read committed）和可重复读（Repeatable Read）。

2.Read uncommitted由于存在脏读，即能读到未提交事务的数据行，所以不适用MVCC。

原因是MVCC的创建版本和删除版本只要在事务提交后才会产生。

3.串行化由于是会对所涉及到的表加锁，并非行锁，自然也就不存在行的版本控制问题。

4.通过以上总结，可知，MVCC主要作用于事务性的，有行锁控制的数据库模型。

## 快照读与当前读

### 1. 快照读

使用 MVCC 读取的是快照中的数据，这样可以减少加锁所带来的开销。

```
select * from table ...;
```

### 2. 当前读

读取的是最新的数据，需要加锁。以下第一个语句需要加 S 锁，其它都需要加 X 锁。

```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert;
update;
delete;
```

## Next-Key Locks

Next-Key Locks 是 MySQL 的 InnoDB 存储引擎的一种锁实现。

MVCC 不能解决幻影读问题，Next-Key Locks 就是为了解决这个问题而存在的。在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

## Record Locks

锁定一个记录上的索引，而不是记录本身。

如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚簇索引，因此 Record Locks 依然可以使用。

## Gap Locks

锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就不能在 t.c 中插入 15。

```
SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE;
```

## Next-Key Locks

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录上的索引，也锁定索引之间的间隙。例如一个索引包含以下值：10, 11, 13, and 20，那么就需要锁定以下区间：

```
(-∞, 10]
(10, 11]
(11, 13]
(13, 20]
(20, +∞)
```

## 范式

### 1NF(第一范式)

第一范式是指数据库表中的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

简而言之，第一范式就是无重复的列。例如，由“职工号”“姓名”“电话号码”组成的表(一个人可能有一部办公电话和一部移动电话)，这时将其规范化为1NF可以将电话号码分为“办公电话”和“移动电话”两个属性，即职工(职工号，姓名，办公电话，移动电话)。

### 2NF(第二范式)

第二范式(2NF)是在第一范式(1NF)的基础上建立起来的，即满足第二范式(2NF)必须先满足第一范式(1NF)。第二范式(2NF)要求数据库表中的每个实例或行必须可以被唯一地区分（每个非主属性完全函数依赖于键码）。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。

如果关系模型R为第一范式，并且R中的每一个非主属性完全函数依赖于R的某个候选键，则称R为第二范式模式(如果A是关系模式R的候选键的一个属性，则称A是R的主属性，否则称A是R的非主属性)。

### 3NF(第三范式)

如果关系模型R是第二范式，且每个非主属性都不传递依赖于R的候选键，则称R是第三范式的模式。

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

存在以下传递函数依赖，不符合3NF：

- Sno  $\rightarrow$  Sdept  $\rightarrow$  Mname

### BCNF(BC范式)

它构建在第三范式的基础上，如果关系模型R是第一范式，且每个属性都不传递依赖于R的候选键，那么称R为BCNF的模式。

即存在关键字段决定关键字段的情况，则其不符合BCNF。

### 4NF(第四范式)

设R是一个关系模型，D是R上的多值依赖集合。如果D中存在凡多值依赖X->Y时，X必是R的超键，那么称R是第四范式的模式。

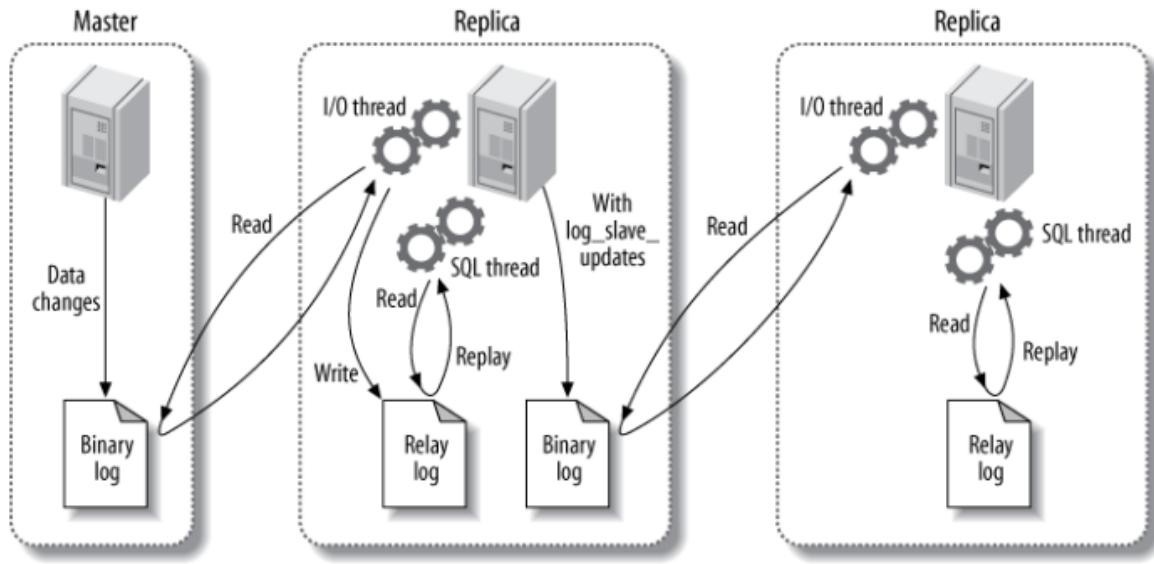
例如，职工表(职工编号，职工孩子姓名，职工选修课程)，在这个表中，同一个职工可能会有多个职工孩子姓名，同样，同一个职工可能会有多个职工选修课程，即这里存在着多值事实，不符合第四范式。如果要符合第四范式，只需要将上表分为两个表，使它们只有一个多值事实，例如职工表一(职工编号，职工孩子姓名)，职工表二(职工编号，职工选修课程)，两个表都只有一个多值事实，所以符合第四范式。

## MySQL主从复制和读写分离

### 主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

- **binlog 线程**：负责将主服务器上的数据更改写入二进制日志 (Binary log) 中。
- **I/O 线程**：负责从主服务器上读取二进制日志，并写入从服务器的中继日志 (Relay log)。
- **SQL 线程**：负责读取中继日志，解析出主服务器已经执行的数据更改并在从服务器中重放 (Replay)。



## MySQL 主从同步延时问题 (精华)

以前线上确实处理过因为主从同步延时问题而导致的线上的 bug，属于小型的生产事故。

是这个么场景。有个同学是这样写代码逻辑的。先插入一条数据，再把它查出来，然后更新这条数据。在生产环境高峰期，写并发达到了 2000/s，这个时候，主从复制延时大概是在小几十毫秒。线上会发现，每天总有那么一些数据，我们期望更新一些重要的数据状态，但在高峰期时候却没更新。用户跟客服反馈，而客服就会反馈给我们。

我们通过 MySQL 命令：

```
show status
```

查看 `Seconds_Behind_Master`，可以看到从库复制主库的数据落后了几 ms。

一般来说，如果主从延迟较为严重，有以下解决方案：

- 分库，将一个主库拆分为多个主库，每个主库的写并发就减少了几倍，此时主从延迟可以忽略不计。
- 打开 MySQL 支持的并行复制，多个库并行复制。如果说某个库的写入并发就是特别高，单库写并发达到了 2000/s，平行复制还是没意义。
- 重写代码，写代码的同学，要慎重，插入数据时立马查询可能查不到。
- 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询 **设置直连主库。不推荐这种方法**，你要是这么搞，读写分离的意义就丧失了。

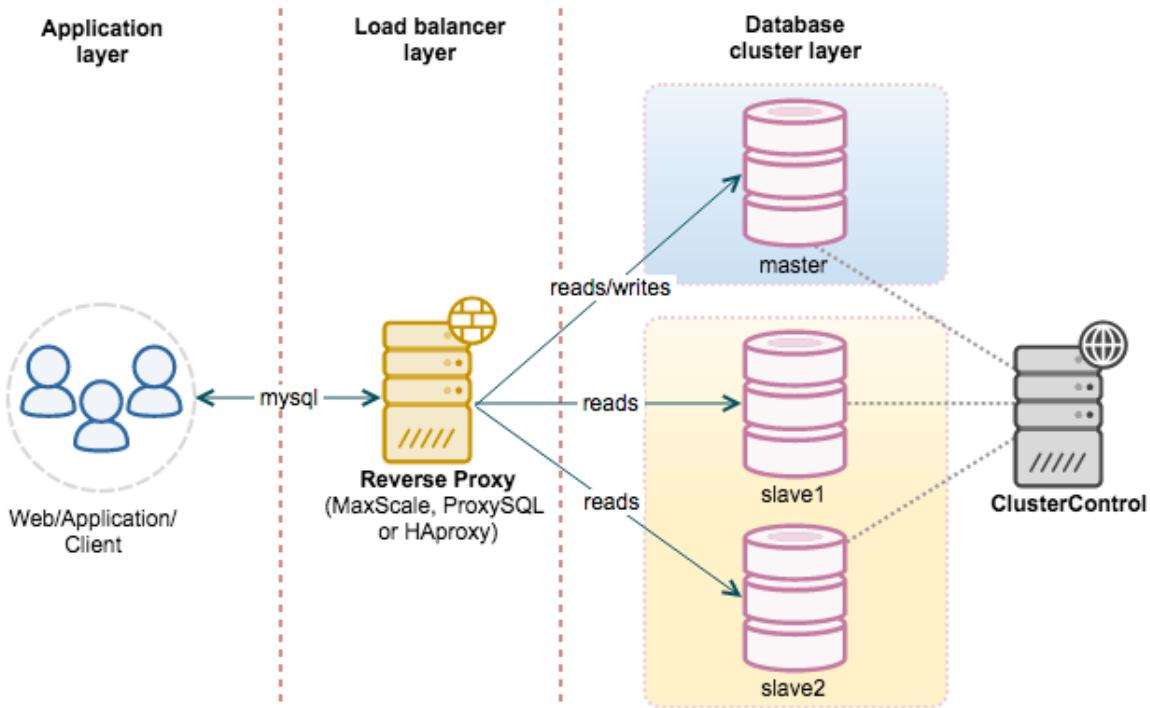
## 读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

- 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
- 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；
- 增加冗余，提高可用性。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。



## 索引的使用条件

- 对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效；
- 对于中到大型的表，索引就非常有效；
- 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

## 索引优化

### 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用 actor\_id 列的索引：

```
SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

### 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor\_id 和 film\_id 设置为多列索引。

```
SELECT film_id, actor_id FROM sakila.film_actor
WHERE actor_id = 1 AND film_id = 1;
```

### 索引列的顺序

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，每个记录的区分度越高，查询效率也越高。

例如下面显示的结果中 customer\_id 的选择性比 staff\_id 更高，因此最好把 customer\_id 列放在多列索引的前面。

```
SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
COUNT(*)
FROM payment;
staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
COUNT(*): 16049
```

## 最左匹配原则

- MySQL会一直向右匹配直到遇到范围查询 (>、<、between、like) 就停止匹配，比如a=3 and b=4 and c>5 and d=6，如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引，则是都可以用到，a、b和d的顺序可以任意调整
- =和in可以乱序，比如a=1 and b=2 and c=3建立(a,b,c)索引可以任意顺序，MySQL的查询优化器会优化成索引可识别的形式

## 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

## 覆盖索引

索引包含所有需要查询的字段的值。

具有以下优点：

- 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
- 一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
- 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

## 索引的优点

- 大大减少了服务器需要扫描的数据行数。
- 帮助服务器避免进行排序和分组，以及避免创建临时表（B+Tree 索引是有序的，可以用于 ORDER BY 和 GROUP BY 操作。临时表主要是在排序和分组过程中创建，不需要排序和分组，也就不需要创建临时表）。
- 将随机 I/O 变为顺序 I/O（B+Tree 索引是有序的，会将相邻的数据都存储在一起）。

## 数据库 SQL 开发规范

### 1. 建议使用预编译语句进行数据库操作

预编译语句可以重复使用这些计划，减少 SQL 编译所需要的时间，还可以解决动态 SQL 所带来的 SQL 注入的问题。

只传参数，比传递 SQL 语句更高效。

相同语句可以一次解析，多次使用，提高处理效率。

### 2. 避免数据类型的隐式转换

隐式转换会导致索引失效如：

```
select name,phone from customer where id = '111';
```

### 3. 充分利用表上已经存在的索引

避免使用双%号的查询条件。如：`a like '%123%'`，（如果无前置%，只有后置%，是可以用到列上的索引的）

一个 SQL 只能利用到复合索引中的一列进行范围查询。如：有 a,b,c 列的联合索引，在查询条件中有 a 列的范围查询，则在 b,c 列上的索引将不会被用到。

在定义联合索引时，如果 a 列要用到范围查找的话，就要把 a 列放到联合索引的右侧，使用 left join 或 not exists 来优化 not in 操作，因为 not in 也通常会使用索引失效。

### 4. 数据库设计时，应该要对以后扩展进行考虑

### 5. 程序连接不同的数据库使用不同的账号，禁止跨库查询

- 为数据库迁移和分库分表留出余地
- 降低业务耦合度
- 避免权限过大而产生的安全风险

### 6. 禁止使用 SELECT \* 必须使用 SELECT <字段列表> 查询

原因：

- 消耗更多的 CPU 和 IO 以网络带宽资源
- 无法使用覆盖索引
- 可减少表结构变更带来的影响

### 7. 禁止使用不含字段列表的 INSERT 语句

如：

```
insert into values ('a','b','c');
```

应使用：

```
insert into t(c1,c2,c3) values ('a','b','c');
```

### 8. 避免使用子查询，可以把子查询优化为 JOIN 操作

通常子查询在 in 子句中，且子查询中为简单 SQL(不包含 union、group by、order by、limit 从句) 时，才可以把子查询转化为关联查询进行优化。

**子查询性能差的原因：**

子查询的结果集无法使用索引，通常子查询的结果集会被存储到临时表中，不论是内存临时表还是磁盘临时表都不会存在索引，所以查询性能会受到一定的影响。特别是对于返回结果集比较大的子查询，其对查询性能的影响也就越大。

由于子查询会产生大量的临时表也没有索引，所以会消耗过多的 CPU 和 IO 资源，产生大量的慢查询。

### 9. 避免使用 JOIN 关联太多的表

对于 MySQL 来说，是存在关联缓存的，缓存的大小可以由 `join_buffer_size` 参数进行设置。

在 MySQL 中，对于同一个 SQL 多关联 (join) 一个表，就会多分配一个关联缓存，如果在一个 SQL 中关联的表越多，所占用的内存也就越大。

如果程序中大量的使用了多表关联的操作，同时 join\_buffer\_size 设置的也不合理的情况下，就容易造成服务器内存溢出的情况，就会影响到服务器数据库性能的稳定性。

同时对于关联操作来说，会产生临时表操作，影响查询效率，MySQL 最多允许关联 61 个表，建议不超过 5 个。

## 10. 减少同数据库的交互次数

数据库更适合处理批量操作，合并多个相同的操作到一起，可以提高处理效率。

## 11. 对应同一列进行 or 判断时，使用 in 替代 or

in 的值不要超过 500 个，in 操作可以更有效的利用索引，or 大多数情况下很少能利用到索引。

## 12. 禁止使用 order by rand() 进行随机排序

order by rand() 会把表中所有符合条件的数据装载到内存中，然后在内存中对所有数据根据随机生成的值进行排序，并且可能会对每一行都生成一个随机值，如果满足条件的数据集非常大，就会消耗大量的 CPU 和 IO 及内存资源。

推荐在程序中获取一个随机值，然后从数据库中获取数据的方式。

## 13. WHERE 从句中禁止对列进行函数转换和计算

对列进行函数转换或计算时会导致无法使用索引

不推荐：

```
where date(create_time)='20190101'
```

推荐：

```
where create_time >= '20190101' and create_time < '20190102'
```

## 14. 在明显不会有重复值时使用 UNION ALL 而不是 UNION

- UNION 会把两个结果集的所有数据放到临时表中后再进行去重操作
- UNION ALL 不会再对结果集进行去重操作

## 15. 拆分复杂的大 SQL 为多个小 SQL

- 大 SQL 逻辑上比较复杂，需要占用大量 CPU 进行计算的 SQL
- MySQL 中，一个 SQL 只能使用一个 CPU 进行计算
- SQL 拆分后可以通过并行执行来提高处理效率

# 分页优化

```
select * from table_name limit 10000,10
```

这句 SQL 的执行逻辑是：

1. 从数据表中读取第N条数据添加到数据集中

2. 重复第一步直到  $N = 10000 + 10$
3. 根据 offset 抛弃前面 10000 条数
4. 返回剩余的 10 条数据

数据库的数据存储是随机的，使用 B+Tree, Hash 等方式组织索引。所以当你让数据库读取第 10001 条数据的时候，数据库就只能一条一条的去查去数

最简单的方法是利用自增索引（假设为 id）：

```
select * from table_name where (id >= 10000) limit 10
```

但思路是有局限性的，首先必须要有自增索引列，而且数据在逻辑上必须是连续的，其次，你还必须知道特征值。如此苛刻的要求，在实际应用中是不可能满足的

好的方式是利用子查询把原来的基于 user 的搜索转化为基于主键 (id) 的搜索，主查询因为已经获得了准确的索引值，所以查询过程也相对较快：

```
select * from table_name inner join ( select id from table_name where (user = xxx) limit 10000,10 ) b using (id)
```

## 存储引擎

### InnoDB

是 MySQL 默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ 间隙锁（Next-Key Locking）防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

若一个主键被定义，该主键作为聚集索引；若没有主键被定义，该表的第一个唯一非空索引则作为聚集索引；若均不满足，InnoDB 内部会生成一个隐藏主键（聚集索引）。非主键索引存储相关键位和其对应的主键值，包含两次查找

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

### MyISAM

设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了 `DELAY_KEY_WRITE` 选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

## 比较

- 事务：InnoDB 是事务型的，可以使用 `Commit` 和 `Rollback` 语句。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键：InnoDB 支持外键。
- 备份：InnoDB 支持在线热备份。
- 崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

## 适用场景

### MyISAM

- 适合频繁执行全表 `count` 语句
- 适合对数据进行增删改的频率不高，查询非常频繁
- 没有事务

### InnoDB

- 数据增删改查都相当频繁
- 可靠性要求比较高，要求支持事务

## MySQL之SQL语句优化步骤

### MySQL查询截取分析步骤：

- 一、开启慢查询日志，捕获慢SQL
- 二、`explain`+慢SQL分析
- 三、`show profile`查询SQL语句在服务器中的执行细节和生命周期
- 四、SQL数据库服务器参数调优

### 一、开启慢查询日志，捕获慢SQL

#### 1. 查看慢查询日志是否开启

```
SHOW VARIABLES LIKE '%slow_query_log';
```

```
mysql> SHOW VARIABLES LIKE '%slow_query_log%';
+-----+-----+
| Variable_name      | Value
+-----+-----+
| slow_query_log     | OFF
| slow_query_log_file | /usr/local/mysql/data/hua-slow.log |
+-----+-----+
2 rows in set (0.00 sec)          http://blog.csdn.net/DrDanger
```

#### 2. 开启慢查询日志

```
SET GLOBAL slow_query_log=1;
```

```
mysql> set global slow_query_log=1;
Query OK, 0 rows affected (0.12 sec)

mysql> SHOW VARIABLES LIKE '%slow_query_log%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| slow_query_log | ON |
| slow_query_log_file | /usr/local/mysql/data/hua-slow.log |
+-----+-----+
2 rows in set (0.00 sec) http://blog.csdn.net/DrDanger
```

### 3、查看慢查询日志阈值

```
SHOW [GLOBAL] VARIABLES LIKE '%long_query_time%';
```

这个值表示超过多长时间的SQL语句会被记录到慢查询日志中

```
mysql> SHOW GLOBAL VARIABLES LIKE '%long_query_time%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| long_query_time | 10.000000 |
+-----+-----+
1 row in set (0.00 sec) http://blog.csdn.net/DrDanger
```

### 4、设置慢查询日志阈值

```
SET GLOBAL long_query_time=3;
```

```
mysql> set global long_query_time=3;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW GLOBAL VARIABLES LIKE '%long_query_time%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| long_query_time | 3.000000 |
+-----+-----+
1 row in set (0.00 sec) http://blog.csdn.net/DrDanger
```

### 5、查看多少SQL语句超过了阈值

```
SHOW GLOBAL STATUS LIKE '%slow_queries%';
```

```

mysql> SHOW GLOBAL STATUS LIKE '%Slow_queries%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Slow_queries | 0    |
+-----+-----+
1 row in set (0.00 sec)          http://blog.csdn.net/DrDanger

```

## 6、MySQL提供的日志分析工具mysqldumpslow

进入MySQL的安装目录中的bin目录下

```

[root@hua ~]# cd /usr/local/mysql/
[root@hua mysql]# ll
总用量 76
drwxr-xr-x. 2 root  root  4096 1月   8 11:00 bin
-rw-r--r--. 1 root  root  17987 9月  13 23:49 COPYING
drwxr-xr-x. 6 mysql mysql 4096 1月  18 09:16 data
drwxr-xr-x. 2 root  root  4096 1月   8 10:59 docs
drwxr-xr-x. 3 root  root  4096 1月   8 10:59 include
drwxr-xr-x. 3 root  root  4096 1月   8 11:00 lib
drwxr-xr-x. 4 root  root  4096 1月   8 11:00 man
-rw-r--r--. 1 root  root   943 1月   8 11:25 my.cnf
-rw-r--r--. 1 root  root   943 1月   8 11:25 my-new.cnf
drwxr-xr-x. 10 root root 4096 1月   8 10:59 mysql-test
-rw-r--r--. 1 root  root  2496 9月  13 23:49 README
drwxr-xr-x. 2 root  root  4096 1月   8 11:00 scripts
drwxr-xr-x. 28 root root 4096 1月   8 10:59 share
drwxr-xr-x. 4 root  root  4096 1月   8 10:59 sql-bench
drwxr-xr-x. 2 root  root  4096 1月   8 11:00 support-files
[root@hua mysql]# cd bin/          http://blog.csdn.net/DrDanger

```

执行 ./mysqldumpslow --help 查看帮助命令

常用参考：

得到返回记录集最多的10个SQL

mysqldumpslow -s r -t 10 slow.log

得到访问次数最多的10个SQL

mysqldumpslow -s c -t 10 slow.log

得到按照时间排序的前10条里面含有左连接的查询语句

mysqldumpslow -s t -t 10 -g "left join" slow.log

使用这些语句时结合 | more 使用

## 二、explain+慢SQL分析

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈。

使用方式：Explain+SQL语句

执行计划包含的信息：

```

+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type  | possible_keys | key   | key_len | ref   | rows  | Extra  |
+-----+-----+-----+-----+-----+-----+-----+

```

### 1、id

SELECT查询的序列号，包含一组数字，表示查询中执行SELECT语句或操作表的顺序

包含三种情况：

- 1.id相同，执行顺序由上至下
- 2.id不同，如果是子查询，id序号会递增，id值越大优先级越高，越先被执行
- 3.id既有相同的，又有不同的。id如果相同认为是一组，执行顺序由上至下；在所有组中，id值越大优先级越高，越先执行。

## 2、**select\_type**

**SIMPLE**:简单SELECT查询，查询中不包含子查询或者UNION

**PRIMARY**:查询中包含任何复杂的子部分，最外层的查询

**SUBQUERY**: SELECT或WHERE中包含的子查询部分

**DERIVED**: 在FROM中包含的子查询被标记为DERIVED(衍生)，MySQL会递归执行这些子查询，把结果放到临时表中

**UNION**: 若第二个SELECT出现UNION，则被标记为UNION, 若UNION包含在FROM子句的子查询中，外层子查询将被标记为DERIVED

**UNION RESULT**: 从UNION表获取结果的SELECT

## 3、**table**

显示这一行数据是关于哪张表的

## 4、**type**

type显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system>const>eq\_ref>ref>fulltext>ref\_or\_null>index\_merge>unique\_subquery>index\_subquery>range>index>ALL

一般来说，得保证查询至少达到range级别，最好能达到ref。

**system**: 表只有一行记录（等于系统表），这是const类型的特例，平时不会出现

**const**: 如果通过索引依次就找到了，const用于比较主键索引或者unique索引。因为只能匹配一行数据，所以很快。如果将主键置于where列表中，MySQL就能将该查询转换为一个常量

**eq\_ref**: 唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描

**ref**: 非唯一性索引扫描，返回匹配某个单独值的所有行。本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而它可能会找到多个符合条件的行，所以它应该属于查找和扫描的混合体

**range**: 只检索给定范围的行，使用一个索引来选择行。key列显示使用了哪个索引，一般就是在你的where语句中出现between、<、>、in等的查询，这种范围扫描索引比全表扫描要好，因为只需要开始于缩印的某一点，而结束于另一点，不用扫描全部索引

**index**: Full Index Scan , index与ALL的区别为index类型只遍历索引树，这通常比ALL快，因为索引文件通常比数据文件小。（也就是说虽然ALL和index都是读全表，但index是从索引中读取的，而ALL是从硬盘读取的）

**all**: Full Table Scan，遍历全表获得匹配的行

## 5、**possible\_keys**

显示可能应用在这张表中的索引，一个或多个。查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用

## 6、**key**

实际使用的索引。如果为NULL，则没有使用索引。

查询中若出现了覆盖索引，则该索引仅出现在key列表中。

## 7、key\_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精度的情况下，长度越短越好。

key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的。

## 8、ref

显示索引的哪一列被使用了，哪些列或常量被用于查找索引列上的值。

## 9、rows

根据表统计信息及索引选用情况，大致估算出找到所需记录多需要读取的行数。

## 10、Extra

包含不适合在其他列中显示但十分重要的额外信息：

- 1、Using filesort：说明MySQL会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。MySQL中无法利用索引完成的排序操作称为“文件排序”
- 2、Using temporary：使用了临时表保存中间结果，MySQL在对查询结果排序时使用临时表。常见于排序order by和分组查询group by
- 3、Using index：表示相应的SELECT操作中使用了覆盖索引（Covering Index），避免访问了表的数据行，效率不错。如果同时出现using where，表明索引被用来执行索引键值的查找；如果没有同时出现using where，表明索引用来读取数据而非执行查找动作 覆盖索引（Covering Index）：理解方式1：SELECT的数据列只需要从索引中就能读取到，不需要读取数据行，MySQL可以利用索引返回SELECT列表中的字段，而不必根据索引再次读取数据文件，换句话说查询列要被所建的索引覆盖 理解方式2：索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此他不必读取整个行。毕竟索引叶子节点存储了他们索引的数据；当能通过读取索引就可以得到想要的数据，那就不需要读取行了，一个索引包含了（覆盖）满足查询结果的数据就叫做覆盖索引 注意：如果要使用覆盖索引，一定要注意SELECT列表中只取出需要的列，不可SELECT \*，因为如果所有字段一起做索引会导致索引文件过大查询性能下降
- 6、impossible where：WHERE子句的值总是false，不能用来获取任何元组
- 7、select tables optimized away：在没有GROUP BY子句的情况下基于索引优化MIN/MAX操作或者对于MyISAM存储引擎优化COUNT(\*)操作，不必等到执行阶段再进行计算，查询执行计划生成的阶段即完成优化
- 8、distinct：优化distinct操作，在找到第一匹配的元祖后即停止找同样值的操作

## 三、show profile查询SQL语句在服务器中的执行细节和生命周期

Show Profile是MySQL提供可以用来分析当前会话中语句执行的资源消耗情况，可以用于SQL的调优测量

默认关闭，并保存最近15次的运行结果

分析步骤

- 1、查看状态：SHOW VARIABLES LIKE 'profiling';
- 2、开启：set profiling=on;
- 3、查看结果：show profiles;
- 4、诊断SQL：show profile cpu,block io for query 上一步SQL数字号码;  
ALL：显示所有开销信息  
BLOCK IO：显示IO相关开销  
CONTEXT SWITCHES：显示上下文切换相关开销  
CPU：显示CPU相关开销  
IPC：显示发送接收相关开销  
MEMORY：显示内存相关开销

PAGE FAULTS: 显示页面错误相关开销  
SOURCE: 显示和Source\_function, Source\_file, Source\_line相关开销  
SWAPS: 显示交换次数相关开销  
注意 (遇到这几种情况要优化)  
converting HEAP to MyISAM: 查询结果太大, 内存不够用往磁盘上搬  
Creating tmp table: 创建临时表  
Copying to tmp table on disk: 把内存中的临时表复制到磁盘  
locked

## 四、SQL数据库服务器参数调优

当order by 和 group by无法使用索引时, 增大max\_length\_for\_sort\_data参数设置和增大sort\_buffer\_size参数的设置

## 悲观锁和乐观锁

悲观锁: select for update

乐观锁: 先查询一次数据, 然后使用查询出来的数据+1进行更新数据, 如果失败则循环

## 优化数据访问

### 1. 减少请求的数据量

- 只返回必要的列: 最好不要使用 SELECT \* 语句。
- 只返回必要的行: 使用 LIMIT 语句来限制返回的数据。
- 缓存重复查询的数据: 使用缓存可以避免在数据库中进行查询, 特别在要查询的数据经常被重复查询时, 缓存带来的查询性能提升将会是非常明显的。

### 2. 减少服务器端扫描的行数

最有效的方式是使用索引来覆盖查询。

## mysql表在磁盘中的存储方式

数据库中的数据都存储在表空间中。表空间相当于是从数据库的逻辑存储到磁盘物理存储的一个映射, 用于指明数据的物理位置。

一般一个数据库对应若干个表空间, 一个表空间对应多个段, 一个段对应多个区, 一个区对应多个数据块, 数据就保存在数据块中。如果再往深层次追究每次查找数据是怎样从磁盘读取的, 就要考虑B+树索引的相关内容。

## SQL语句分析

### 查询语句

说了以上这么多, 那么究竟一条sql语句是如何执行的呢? 其实我们的sql可以分为两种, 一种是查询, 一种是更新(增加, 更新, 删除)。我们先分析下查询语句, 语句如下:

```
select * from tb_student A where A.age='18' and A.name='张三';
```

结合上面的说明, 我们分析下这个语句的执行流程:

- 先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限，在 MySQL8.0 版本以前，会先查询缓存，以这条 sql 语句为 key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。
- 通过分析器进行词法分析，提取 sql 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb\_student，需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 sql 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。
- 接下来就是优化器进行确定执行方案，上面的 sql 语句，可以有两种执行方案：

- 先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。
- 先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。

那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。

- 进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

## 更新语句

以上就是一条查询 sql 的执行流程，那么接下来我们看看一条更新语句如何执行的呢？sql 语句如下：

```
update tb_student A set A.age='19' where A.name=' 张三 ';
```

我们来给张三修改下年龄，在实际数据库肯定不会设置年龄这个字段的，不然要被技术负责人打的。其实这条语句也基本上会沿着上一个查询的流程走，只不过执行更新的时候肯定要记录日志啦，这就会引入日志模块了，MySQL 自带的日志模块式 **binlog (归档日志)**，所有的存储引擎都可以使用，我们常用的 InnoDB 引擎还自带了一个日志模块 **redo log (重做日志)**，我们就以 InnoDB 模式下来探讨这个语句的执行流程。流程如下：

- 先查询到张三这一条数据，如果有缓存，也是会用到缓存。
- 然后拿到查询的语句，把 age 改为 19，然后调用引擎 API 接口，写入这一行数据，InnoDB 引擎把数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。
- 执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。
- 更新完成。

### 这里肯定有同学会问，为什么要用两个日志模块，用一个日志模块不行吗？

这是因为最开始 MySQL 并没与 InnoDB 引擎（InnoDB 引擎是其他公司以插件形式插入 MySQL 的），MySQL 自带的引擎是 MyISAM，但是我们知道 redo log 是 InnoDB 引擎特有的，其他存储引擎都没有，这就导致会没有 crash-safe 的能力（crash-safe 的能力即使数据库发生异常重启，之前提交的记录都不会丢失），binlog 日志只能用来归档。

并不是说只用一个日志模块不可以，只是 InnoDB 引擎就是通过 redo log 来支持事务的。那么，又会有同学问，我用两个日志模块，但是不要这么复杂行不行，为什么 redo log 要引入 prepare 预提交状态？这里我们用反证法来说明下为什么要这么做？

- 先写 redo log 直接提交，然后写 binlog**，假设写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 binlog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。
- 先写 binlog，然后写 redo log**，假设写完了 binlog，机器异常重启了，由于没有 redo log，本机是无法恢复这一条记录的，但是 binlog 又有记录，那么和上面同样的道理，就会产生数据不一致的情况。

如果采用 redo log 两阶段提交的方式就不一样了，写完 binlog 后，然后再提交 redo log 就会防止出现上述的问题，从而保证了数据的一致性。那么问题来了，有没有一个极端的情况呢？假设 redo log 处于预提交状态，binlog 也已经写完了，这个时候发生了异常重启会怎么样呢？这个就要依赖于 MySQL 的处理机制了，MySQL 的处理过程如下：

- 判断 redo log 是否完整，如果判断是完整的，就立即提交。
- 如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log，不完整就回滚事务。

这样就解决了数据一致性的问题。

## 连接

连接用于连接多个表，使用 JOIN 关键字，并且条件语句使用 ON 而不是 WHERE。

连接可以替换子查询，并且比子查询的效率一般会更快。

可以用 AS 给列名、计算字段和表名取别名，给表名取别名是为了简化 SQL 语句以及连接相同表。

### 内连接

内连接又称等值连接，使用 INNER JOIN 关键字。

```
SELECT A.value, B.value
FROM tablea AS A INNER JOIN tableb AS B
ON A.key = B.key;
```

可以不明确使用 INNER JOIN，而使用普通查询并在 WHERE 中将两个表中要连接的列用等值方法连接起来。

```
SELECT A.value, B.value
FROM tablea AS A, tableb AS B
WHERE A.key = B.key;
```

### 自连接

自连接可以看成内连接的一种，只是连接的表是自身而已。

一张员工表，包含员工姓名和员工所属部门，要找出与 Jim 处在同一部门的所有员工姓名。

子查询版本

```
SELECT name
FROM employee
WHERE department = (
    SELECT department
    FROM employee
    WHERE name = "Jim");
```

自连接版本

```
SELECT e1.name
FROM employee AS e1 INNER JOIN employee AS e2
ON e1.department = e2.department
AND e2.name = "Jim";
```

## 自然连接

自然连接是把同名列通过等值测试连接起来的，同名列可以有多个。

内连接和自然连接的区别：内连接提供连接的列，而自然连接自动连接所有同名列。

```
SELECT A.value, B.value  
FROM tablea AS A NATURAL JOIN tableb AS B;
```

## 外连接

外连接保留了没有关联的那些行。分为左外连接，右外连接以及全外连接，左外连接就是保留左表没有关联的行。

检索所有顾客的订单信息，包括还没有订单信息的顾客。

```
SELECT Customers.cust_id, Orders.order_num  
FROM Customers LEFT OUTER JOIN Orders  
ON Customers.cust_id = Orders.cust_id;
```

customers 表：

cust_id	cust_name
1	a
2	b
3	c

orders 表：

order_id	cust_id
1	1
2	1
3	3
4	3

结果：

cust_id	cust_name	order_id
1	a	1
1	a	2
3	c	3
3	c	4
2	b	Null

# MongoDB使用B树， MySQL使用B+树

- MongoDB 是一种 nosql，也存储在磁盘上，被设计用在 数据模型简单，性能要求高的场合。性能要求高，看看B/B+树的区别第一点：

B+树内节点不存储数据，所有 data 存储在叶节点导致查询时间复杂度固定为  $\log n$ 。而B-树查询时间复杂度不固定，与 key 在树中的位置有关，最好为O(1)

我们说过，尽可能少的磁盘 IO 是提高性能的有效手段。MongoDB 是聚合型数据库，而 B-树恰好 key 和 data 域聚合在一起。

- Mysql 是一种关系型数据库，区间访问是常见的一种情况，而 B-树并不支持区间访问（可参见上图），而B+树由于数据全部存储在叶子节点，并且通过指针串在一起，这样就很容易的进行区间遍历甚至全部遍历。

见B/B+树的区别第二点：

B+树叶节点两两相连可大大增加区间访问性，可使用在范围查询等，而B-树每个节点 key 和 data 在一起，则无法区间查找。

其次B+树的查询效率更加稳定，数据全部存储在叶子节点，查询时间复杂度固定为  $O(\log n)$ 。

最后第三点：

B+树更适合外部存储。由于内节点无 data 域，每个节点能索引的范围更大更精确

## SQL和NoSQL的区别

### 概念

SQL (Structured Query Language) 数据库，指关系型数据库。主要代表：SQL Server，Oracle，MySQL(开源)，PostgreSQL(开源)。

NoSQL (Not Only SQL) 泛指非关系型数据库。主要代表：MongoDB，Redis，CouchDB。

### 区别

#### 1、存储方式

SQL数据存在特定结构的表中；而NoSQL则更加灵活和可扩展，存储方式可以省是JSON文档、哈希表或者其他方式。

#### 2、表/数据集合的数据的关系

在SQL中，必须定义好表和字段结构后才能添加数据，例如定义表的主键(primary key)，索引(index),触发器(trigger),存储过程(stored procedure)等。表结构可以在被定义之后更新，但是如果有比较大的结构变更的话就会变得比较复杂。在NoSQL中，数据可以在任何时候任何地方添加，不需要先定义表。

NoSQL也可以在数据集中建立索引。以MongoDB为例，会自动在数据集合创建后创建唯一值\_id字段，这样的话就可以在数据集创建后增加索引。

从这点来看，NoSQL可能更加适合初始化数据还不明确或者未定的项目中。

#### 3、外部数据存储

SQL中如何需要增加外部关联数据的话，规范化做法是在原表中增加一个外键，关联外部数据表。例如需要在借阅表中增加审核人信息，先建立一个审核人表，再在原来的借阅人表中增加审核人外键。

这样如果我们需要更新审核人个人信息的时候只需要更新审核人表而不需要对借阅人表做更新。而在NoSQL中除了这种规范化的外部数据表做法以外，我们还能用如下的非规范化方式把外部数据直接放到原数据集中，以提高查询效率。缺点也比较明显，更新审核人数据的时候将会比较麻烦。

## 4、SQL中的JOIN查询

SQL中可以使用JOIN表链接方式将多个关系数据表中的数据用一条简单的查询语句查询出来。NoSQL暂未提供类似JOIN的查询方式对多个数据集中的数据做查询。所以大部分NoSQL使用非规范化的数据存储方式存储数据。

## 5、数据耦合性

SQL中不允许删除已经被使用的外部数据，例如审核人表中的“熊三”已经被分配给了借阅人熊大，那么在审核人表中将不允许删除熊三这条数据，以保证数据完整性。而NoSQL中则没有这种强耦合的概念，可以随时删除任何数据。

## 6、事务

SQL中如果多张表数据需要同批次被更新，即如果其中一张表更新失败的话其他表也不能更新成功。这种场景可以通过事务来控制，可以在所有命令完成后再统一提交事务。而NoSQL中没有事务这个概念，每一个数据集的操作都是原子级的。

## 7、增删改查语法

## 8、查询性能

在相同水平的系统设计的前提下，因为NoSQL中省略了JOIN查询的消耗，故理论上性能上是优于SQL的。

## 补充

目前许多大型互联网项目都会选用MySQL（或任何关系型数据库）+ NoSQL的组合方案。

关系型数据库适合存储结构化数据，如用户的帐号、地址：

- 1) 这些数据通常需要做结构化查询（嗯，好像是废话），比如join，这时候，关系型数据库就要胜出一筹
- 2) 这些数据的规模、增长的速度通常是可以预期的
- 3) 事务性、一致性

NoSQL适合存储非结构化数据，如文章、评论：

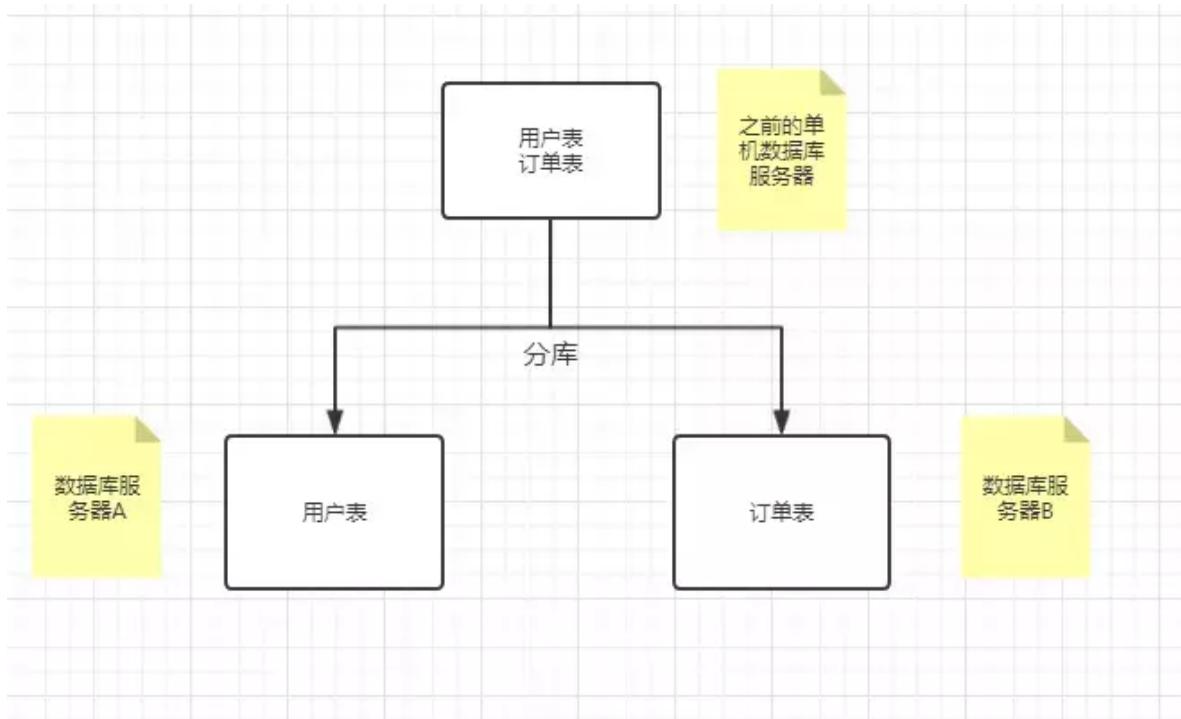
- 1) 这些数据通常用于模糊处理，如全文搜索、机器学习
- 2) 这些数据是海量的，而且增长的速度是难以预期的，
- 3) 根据数据的特点，NoSQL数据库通常具有无限（至少接近）伸缩性
- 4) 按key获取数据效率很高，但是对join或其他结构化查询的支持就比较差

基于它们的适用范围不同，目前主流架构才会采用组合方案，一个也不能少。目前为止，还没有出现一个能够通吃各种场景的数据库，而且根据[CAP理论](#)，这样的数据库是不存在的。

## 分库分表

### 分库

分库讲白了就是比如现在你有一个数据库服务器，数据库中有两张表分别是用户表和订单表。如果要分库的话现在你需要买两台机子，搞两个数据库分别放在两台机子上，并且一个数据库放用户表，一个数据库放订单表，这样存储压力就分担到两个服务器上了，但是会带来新的问题，所以东西变复杂了都会新的问题产生。



1、联表查询问题 也就是join了，之前在一个数据库里面可以用上join用一条sql语句就可以联表查询得到想要的结果，但是现在分为多个数据库了，所以join用不上了。就比如现在要查注册时间在2019年之后用户的订单信息，你就需要先去数据库A中用户表查询注册在2019年之后的信息，然后得到用户id,再拿这些id去数据库B订单表中查找订单信息，然后再拼接这些信息返回。所以等于得多写一些代码了。

2、事务问题 搞数据库基本上都离不开事务，但是现在不同的数据库事务就不是以前那个简单的本地事务了，而是分布式事务了，而引入分布式事务也提高了系统的复杂性，并且有些效率不高还会影响性能例如Mysql XA。还有基于消息中间件实现分布式事务的等等这里不展开讲述。

## 分表

我们已经做了分库了，但是现在情况是我们的表里面的数据太多了，就一不小心你的公司的产品火了，像抖音这种，所有用户如果就存在一张表里吃不消，所以这时候得分表。分别又分垂直分表和水平分表。

### 垂直分表

垂直分表的意思形象点就像坐标轴的y轴，把x轴切成了两半，对应到我们的表就是比如我们表有10列，现在一刀切下去，分成了两张表，其中一张表3列，另一张表7列。

这个一刀切下去让两个表分别有几列不是固定的，垂直分表适合表中存在不常用并且占用了大量空间的表拆分出去。

就拿头条的用户信息，比如用户表只有用户id、昵称、手机号、个人简介这4个字段。但是手机号和个人简介这种信息就属于不太常用的，占用的空间也不小，个人简介有些人写了一坨。所以就把手机号和个人简介这两列拆分出去。

那垂直分表影响就是之前只要一个查询的，现在需要两次查询才能拿到分表之前的完整用户表信息。

### 水平分表

水平分表的意思形象点就像坐标轴的x轴，把y轴切成了两半(当然不仅限于切一刀，可以切好几份)。也拿用户表来说比如现在用户表有5000万行数据，我们切5刀，分成5个表，每个表1000万行数据。

水平分表就适合用户表行数很多的情况下，一般单表行数超过5000万就得分表，如果单表的数据比较复杂那可能2000万甚至1000万就得分了，这个得看实际情况有些表很简单可能一亿行都不用分。所以当一个表行数超过千万级别的时候关注一下，如果没有性能问题就可以再等等看，不要急着分表，因为分表会是带来很多问题。

水平分表的问题比垂直分表就更烦了。

要考虑怎么切，讲的高级点就叫路由

1、按id也就是范围路由，比如id值1999万的放一张表，1000万1999放一张表，一次类推。这个得试的，因为范围分的大了，可能性能还有问题，范围分的小了。。那表不得多死。这种分法的好处就是容易切啊，简单粗暴，以后新增的数据分表都不会影响到之前的数据，之前的数据都不需要移动。

2、哈希路由就是取几列哈希一下看看数据哪个库，比如拿id来做哈希，1500取余8等于4，所以这条记录就放在user\_4这个表中，2011取余8等于3，所以这条记录就放在user\_3中。这种分法好处就是分的很均匀，基本上每个表的数据都差不多，但是以后新增数据又得分表了咋办，以前的数据都得动，比较烦！

3、搞一张表来存储路由关系 还是拿用户表来说，就是弄一个路由表，里面存userId和表编号，表示这个userId是这张user表的的。这种方式也简单，之后又要分表了之后改改路由表，迁移一部分数据。但是这种方法导致每次查询都得查两次，并且如果路由表太大了，那路由表又成为瓶颈了！

## 分库分表引起的问题

1. 事务问题。在执行分库之后，由于数据存储到了不同的库上，数据库事务管理出现了困难。如果依赖数据库本身的分布式事务管理功能去执行事务，将付出高昂的性能代价；如果由应用程序去协助控制，形成程序逻辑上的事务，又会造成编程方面的负担。
2. 跨库跨表的join问题。在执行了分库分表之后，难以避免会将原本逻辑关联性很强的数据划分到不同的表、不同的库上，我们无法join位于不同分库的表，也无法join分表粒度不同的表，结果原本一次查询能够完成的业务，可能需要多次查询才能完成。
3. 额外的数据管理负担和数据运算压力。额外的数据管理负担，最显而易见的就是数据的定位问题和数据的增删改查的重复执行问题，这些都可以通过应用程序解决，但必然引起额外的逻辑运算。

# Redis

## Redis为什么用跳表而不用红黑树

Redis是用跳表来实现有序集合。

Redis 中的有序集合支持的核心操作主要有下面这几个：插入一个数据；删除一个数据；查找一个数据；按照区间查找数据（比如查找值在 [100, 356] 之间的数据）；迭代输出有序序列。

其中，插入、删除、查找以及迭代输出有序序列这几个操作，红黑树也可以完成，时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。

## Redis持久化机制

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后恢复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重要一点就是，Redis支持持久化，而且支持两种不同的持久化操作。

Redis的一种持久化方式叫快照（snapshotting, RDB），另一种方式是只追加文件（append-only file,AOF）。

### 快照（snapshotting）持久化（RDB）

Redis可以通过创建快照来获得存储在内存里面的的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save 900 1          #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命令创建快照。  
save 300 10         #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命令创建快照。  
save 60 10000       #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命令创建快照。
```

优点：全量数据快照，文件小，恢复快

缺点：无法保存最近一次快照之后的数据

### AOF（append-only file）持久化

与快照持久化相比，AOF持久化的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数开启：

```
appendonly yes
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
appendfsync always      #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度  
appendfsync everysec   #每秒钟同步一次, 显示地将多个写命令同步到硬盘  
appendfsync no          #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec 选项，让 Redis 每秒同步一次 AOF 文件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

优点：可读性高，适合保存增量数据，数据不易丢失

缺点：文件体积较大，恢复时间长

## Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项 `aof-use-rdb-preamble` 打开）。

如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点，快速加载同时避免丢失过多的数据。当然缺点也是有的，AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

BGSAVE 做镜像全量持久化，AOF 做增量持久化。

## 数据淘汰策略

可以设置内存最大使用量，当内存使用量超出时，会施行数据淘汰策略。

Redis 具体有 6 种淘汰策略：

策略	描述
volatile-lru	从已设置过期时间的数据集中挑选最近最少使用的数据淘汰
volatile-ttl	从已设置过期时间的数据集中挑选将要过期的数据淘汰
volatile-random	从已设置过期时间的数据集中任意选择数据淘汰
allkeys-lru	从所有数据集中挑选最近最少使用的数据淘汰
allkeys-random	从所有数据集中任意选择数据进行淘汰
noeviction	禁止驱逐数据

作为内存数据库，出于对性能和内存消耗的考虑，Redis 的淘汰算法实际实现上并非针对所有 key，而是抽样一小部分并且从中选出被淘汰的 key。

使用 Redis 缓存数据时，为了提高缓存命中率，需要保证缓存数据都是热点数据。可以将内存最大使用量设置为热点数据占用的内存量，然后启用 allkeys-lru 淘汰策略，将最近最少使用的数据淘汰。

Redis 4.0 引入了 volatile-lfu 和 allkeys-lfu 淘汰策略，LFU 策略通过统计访问频率，将访问频率最少的键值对淘汰。

## 缓存穿透、缓存击穿、缓存雪崩概念及解决方案

### 缓存穿透

## 概念

访问一个不存在的key，缓存不起作用，请求会穿透到DB，流量大时DB会挂掉。

## 解决方案

1. 采用布隆过滤器，使用一个足够大的bitmap，用于存储可能访问的key，不存在的key直接被过滤；
2. 访问key未在DB查询到值，也将空值写进缓存，但可以设置较短过期时间。

## 缓存雪崩

### 概念

大量的key设置了相同的过期时间，导致在缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。

## 解决方案

可以给缓存设置过期时间时加上一个随机值时间，使得每个key的过期时间分布开来，不会集中在同一时刻失效。

## 缓存击穿

### 概念

一个存在的key，在缓存过期的一刻，同时有大量的请求，这些请求都会击穿到DB，造成瞬时DB请求量大、压力骤增。

## 解决方案

在访问key之前，采用SETNX (set if not exists) 来设置另一个短期key来锁住当前key的访问，访问结束再删除该短期key。

## Redis为什么那么快

- 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；
- 2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；
- 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 使用多路I/O复用模型：Redis使用epoll作为I/O多路复用技术的实现，而且Redis将epoll的read、write、close等操作转换成事件，不在网络I/O上耗费太多时间，实现对多个FD的读写，提高性能。

## Redis为什么是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

# redis 的并发竞争问题是什么？如何解决这个问题？了解 redis 事务的 CAS 方案吗？

这个也是线上非常常见的一个问题，就是多客户端同时并发写一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

而且 redis 自己就有天然解决这个问题的 CAS 类的乐观锁方案。

某个时刻，多个系统实例都去更新某个 key。可以基于 zookeeper 实现分布式锁。每个系统通过 zookeeper 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 key，别人都不允许读和写。

你要写入缓存的数据，都是从 mysql 里查出来的，都得写入 mysql 中，写入 mysql 中的时候必须保存一个时间戳，从 mysql 查出来的时候，时间戳也查出来。

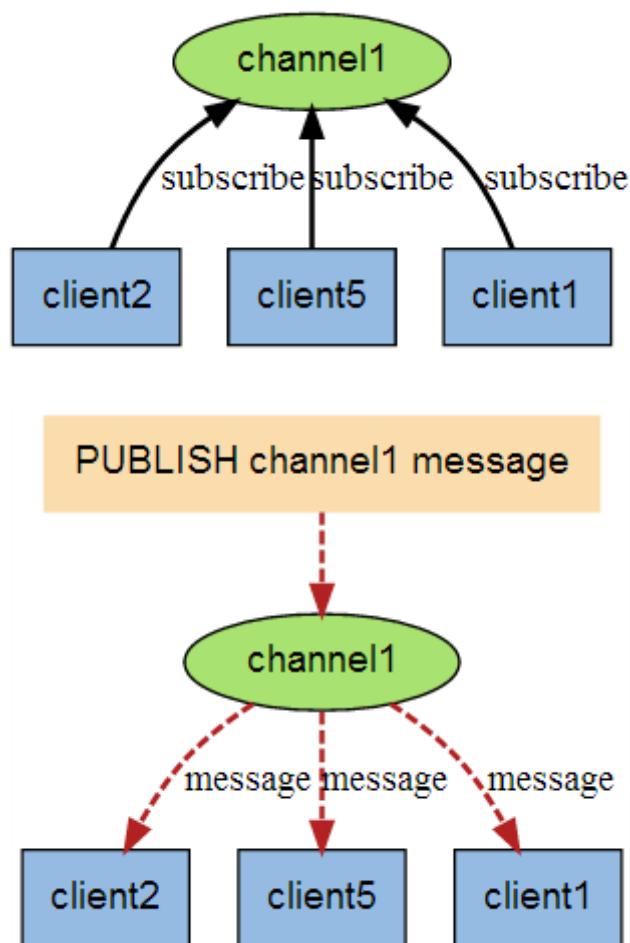
每次要写之前，先判断一下当前这个 value 的时间戳是否比缓存里的 value 的时间戳要新。如果是的话，那么可以写，否则，就不能用旧的数据覆盖新的数据。

## Redis发布订阅

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1，以及订阅这个频道的三个客户端——client2、client5 和 client1 之间的关系：



以下实例演示了发布订阅是如何工作的。在我们实例中我们创建了订阅频道名为 `redisChat`：

```
redis 127.0.0.1:6379> SUBSCRIBE redisChat

Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
```

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息。

```
redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"
(integer) 1

redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by runoob.com"
(integer) 1

# 订阅者的客户端会显示如下消息
1) "message"
2) "redisChat"
3) "Redis is a great caching technique"
1) "message"
2) "redisChat"
3) "Learn redis by runoob.com"
```

消息发布是无状态的，无法保证可达。

## Redis主从同步

### 全同步过程

- master 执行 bgsave，在本地生成一份 rdb 快照文件。
- master node 将 rdb 快照文件发送给 slave node，如果 rdb 复制时间超过 60秒（repl-timeout），那么 slave node 就会认为复制失败，可以适当调大这个参数(对于千兆网卡的机器，一般每秒传输 100MB，6G 文件，很可能超过 60s)
- master node 在生成 rdb 时，会将所有新的写命令缓存在内存中，在 slave node 保存了 rdb 之后，再将新的写命令复制给 slave node。
- 如果在复制期间，内存缓冲区持续消耗超过 64MB，或者一次性超过 256MB，那么停止复制，复制失败。

```
client-output-buffer-limit slave 256MB 64MB 60
```

- slave node 接收到 rdb 之后，清空自己的旧数据，然后重新加载 rdb 到自己的内存中，同时**基于旧的数据版本对外提供服务**。
- 如果 slave node 开启了 AOF，那么会立即执行 BGREWRITEAOF，重写 AOF。

### 增量同步过程

- 如果全量复制过程中，master-slave 网络连接断掉，那么 slave 重新连接 master 时，会触发增量复制。
- master 直接从自己的 backlog 中获取部分丢失的数据，发送给 slave node，默认 backlog 就是 1MB。

- master 就是根据 slave 发送的 psync 中的 offset 来从 backlog 中获取数据的。

## 哨兵

---

### 哨兵功能

- 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

### Redis实现点赞

---

当有用户为一篇文章点赞时，除了要对该文章的 votes 字段进行加 1 操作，还必须记录该用户已经对该文章进行了点赞，防止用户点赞次数超过 1。可以建立文章的已投票用户集合来进行记录。

为了节约内存，规定一篇文章发布满一周之后，就不能再对它进行投票，而文章的已投票集合也会被删除，可以为文章的已投票集合设置一个一周的过期时间就能实现这个规定。

# HTTP

---

## HTTP1.0和HTTP1.1

1. **长连接**: 在HTTP/1.0中，**默认使用的是短连接**，也就是说每次请求都要重新建立一次连接。HTTP是基于TCP/IP协议的，每一次建立或者断开连接都需要三次握手四次挥手的开销，如果每次请求都要这样的话，开销会比较大。因此最好能维持一个长连接，可以用个长连接来发多个请求。**HTTP 1.1起，默认使用长连接**，默认开启Connection: keep-alive。
2. **HTTP/1.1的持续连接有非流水线方式和流水线方式**。流水线方式是客户在收到HTTP的响应报文之前就能接着发送新的请求报文。与之相对应的非流水线方式是客户在收到前一个响应后才能发送下一个请求。
3. **错误状态响应码**: 在HTTP1.1中新增了24个错误状态响应码，如409 (Conflict) 表示请求的资源与资源的当前状态发生冲突；410 (Gone) 表示服务器上的某个资源被永久性的删除。
4. **缓存处理**: 在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag, If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
5. **带宽优化及网络连接的使用**: HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206 (Partial Content)，这样就方便了开发者自由的选择以便充分利用带宽和连接。

## HTTP1.X缺陷

---

HTTP/1.x 实现简单是以牺牲性能为代价的：

- 客户端需要使用多个连接才能实现并发和缩短延迟；
- 不会压缩请求和响应首部，从而导致不必要的网络流量；
- 不支持有效的资源优先级，致使底层 TCP 连接的利用率低下。

## HTTP2.0特性

---

HTTP/2的通过支持请求与响应的多路复用来减少延迟，通过压缩HTTP首部字段将协议开销降至最低，同时增加对请求优先级和服务器端推送的支持。

### (1) 二进制分帧

先来理解几个概念：

**帧**: HTTP/2 数据通信的最小单位消息：指 HTTP/2 中逻辑上的 HTTP 消息。例如请求和响应等，消息由一个或多个帧组成。

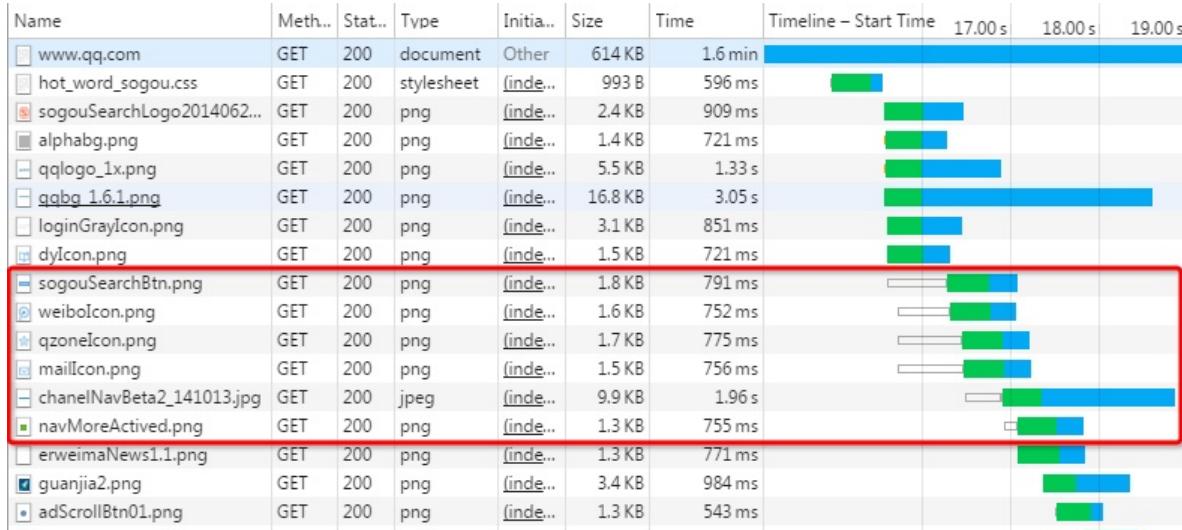
**流**: 存在于连接中的一个虚拟通道。流可以承载双向消息，每个流都有一个唯一的整数ID。

HTTP/2 采用二进制格式传输数据，而非 HTTP 1.x 的文本格式，二进制协议解析起来更高效。HTTP / 1 的请求和响应报文，都是由起始行，首部和实体正文（可选）组成，各部分之间以文本换行符分隔。HTTP/2 将请求和响应数据分割为更小的帧，并且它们采用二进制编码。

**HTTP/2 中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流。**每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装。

## (2) 多路复用

多路复用，代替原来的序列和阻塞机制。所有就是请求的都是通过一个 TCP连接并发完成。HTTP 1.x 中，如果想并发多个请求，必须使用多个 TCP 链接，且浏览器为了控制资源，还会对单个域名有 6-8个的TCP链接请求限制，如下图，红色圈出来的请求就因域名链接数已超过限制，而被挂起等待了一段时间。



在 HTTP/2 中，有了二进制分帧之后，HTTP /2 不再依赖 TCP 链接去实现多流并行了，在 HTTP/2 中：

- 同域名下所有通信都在单个连接上完成。
- 单个连接可以承载任意数量的双向数据流。
- 数据流以消息的形式发送，而消息又由一个或多个帧组成，多个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装。

这一特性，使性能有了极大提升：

- 同个域名只需要占用一个 TCP 连接**，消除了因多个 TCP 连接而带来的延时和内存消耗。
- 单个连接上可以并行交错的请求和响应，之间互不干扰。
- 在 HTTP/2 中，每个请求都可以带一个31bit的优先值，0表示最高优先级，数值越大优先级越低。有了这个优先值，客户端和服务器就可以在处理不同的流时采取不同的策略，以最优的方式发送流、消息和帧。

## (3) 服务器推送

服务端可以在发送页面HTML时主动推送其它资源，而不用等到浏览器解析到相应位置，发起请求再响应。例如服务端可以主动把JS和CSS文件推送给客户端，而不需要客户端解析HTML时再发送这些请求。

服务端可以主动推送，客户端也有权利选择是否接收。如果服务端推送的资源已经被浏览器缓存过，浏览器可以通过发送RST\_STREAM帧来拒收。主动推送也遵守同源策略，服务器不会随便推送第三方资源给客户端。

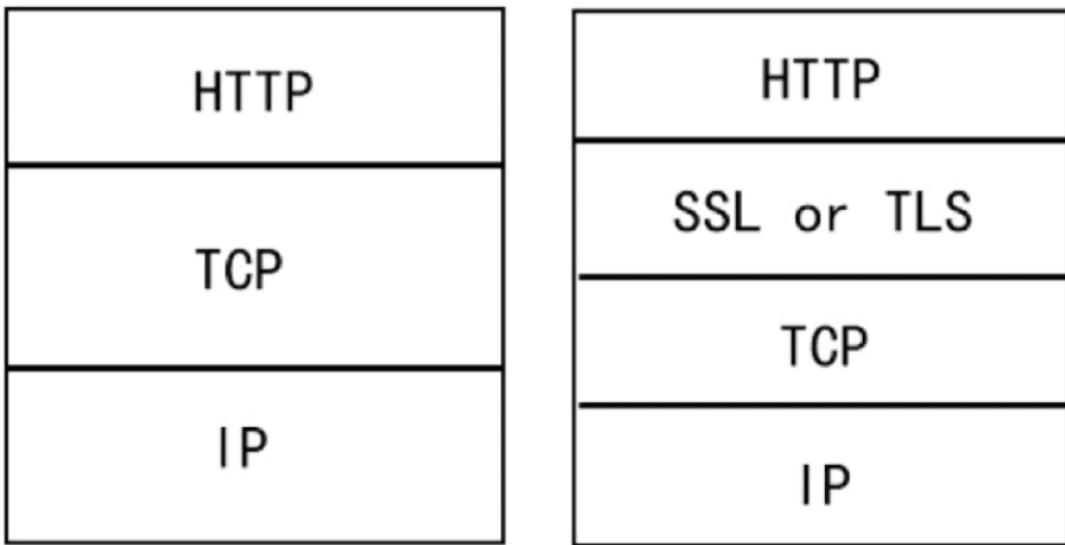
## (4) 头部压缩

HTTP 1.1请求的大小变得越来越大，有时甚至会大于TCP窗口的初始大小，因为它们需要等待带着ACK的响应回来以后才能继续被发送。HTTP/2对消息头采用HPACK（专为http/2头部设计的压缩格式）进行压缩传输，能够节省消息头占用的网络的流量。而HTTP/1.x每次请求，都会携带大量冗余头信息，浪费了很多带宽资源。

# HTTP和HTTPS

## HTTP

## HTTPS



### SSL (Security Socket Layer, 安全套接层)

- 为网络通信提供安全及数据完整性的一种安全协议
- 是操作系统对外的API, SSL3.0后更名为TLS
- 采用身份验证和数据加密保证网络通信的安全和数据的完整性

### 加密的方式

- 对称加密：加密和解密都使用同一个密钥
- 非对称加密：加密使用的密钥和解密使用的密钥是不相同的
- 哈希算法：将任意长度的信息转换为固定长度的值，算法不可逆
- 数字签名：证明某个消息或文件是某人发出/认同的

### HTTPS数据传输流程

- 浏览器将支持的加密算法信息发送给服务器
- 服务器选择一套浏览器支持的加密算法，以证书的形式回发浏览器
- 浏览器验证证书的合法性，并结合证书公钥加密信息发送给服务器
- 服务器使用私钥解密信息，验证哈希，加密相应消息回发浏览器
- 浏览器解密响应消息，并对消息进行验真，之后进行加密交互数据

### 区别

- HTTPS需要到CA申请证书，HTTP不需要
- HTTPS密文传输，HTTP明文传输
- 连接方式不同，HTTPS默认使用443端口，HTTP使用80端口
- HTTPS=HTTP+加密+认证+完整性保护，较HTTP安全

## HTTPS

HTTP 有以下安全性问题：

- 使用明文进行通信，内容可能会被窃听；
- 不验证通信方的身份，通信方的身份有可能遭遇伪装；
- 无法证明报文的完整性，报文有可能遭篡改。

HTTPS 并不是新协议，而是让 HTTP 先和 SSL (Secure Sockets Layer) 通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信。

通过使用 SSL， HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

## HTTPS加密机制

HTTPS 采用混合的加密机制，使用非对称密钥加密用于传输对称密钥来保证传输过程的安全性，之后使用对称密钥加密进行通信来保证通信过程的效率。

## HTTPS加密过程

第一次通信的时候服务端发送公钥给客户端，由客户端产生一个对称密钥，通过服务端的公钥加密后发送给服务端，后续的交互中都通过对称密钥进行加密传输。也就是说先通过非对称密钥加密对称密钥，通过对称密钥加密实际请求的内容。

## HTTPS身份认证过程

服务器提交自己的基本信息向 CA 机构提出申请，CA 机构在给服务器颁发证书的时候，会连同 **数字证书** 以及 **根据证书计算的摘要** 一同发送给服务器，且这个摘要是需要 **经过 CA 机构自己的私钥进行加密的**。

服务器在与客户端通信的时候，就会将数字证书和数字签名出示给客户端了。客户端拿到数字证书和数字签名后，**先通过操作系统或者浏览器内置信任的 CA 机构找到对应 CA 机构的公钥对数字签名进行解密，然后采用同样的摘要算法计算数字证书的摘要，如果自己计算的摘要与服务器发来的摘要一致，则证书是没有被篡改过的！这样就防止了篡改！第三方拿不到 CA 机构的私钥，也就无法对摘要进行加密，如果是第三方伪造的签名自然也在客户端也就无法解密，这就防止了伪造！所以数字签名就是通过这种机制来保证数字证书被篡改和被伪造。**

# GET和POST

## 作用

GET 用于获取资源，而 POST 用于传输实体主体。

## 参数

GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在实体主体中。不能因为 POST 参数存储在实体主体中就认为它的安全性更高，因为照样可以通过一些抓包工具（Fiddler）查看。

因为 URL 只支持 ASCII 码，因此 GET 的参数中如果存在中文等字符就需要先进行编码。例如 中文 会转换为 %E4%B8%AD%E6%96%87，而空格会转换为 %20。POST 参数支持标准字符集。

```
GET /test/demo_form.asp?name1=value1&name2=value2 HTTP/1.1
POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

## 安全

安全的 HTTP 方法不会改变服务器状态，也就是说它只是可读的。

GET 方法是安全的，而 POST 却不是，因为 POST 的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。

安全的方法除了 GET 之外还有：HEAD、OPTIONS。

不安全的方法除了 POST 之外还有 PUT、DELETE。

## 幂等性

幂等的 HTTP 方法，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。换句话说就是，幂等方法不应该具有副作用（统计用途除外）。

所有的安全方法也都是幂等的。

在正确实现的条件下，GET，HEAD，PUT 和 DELETE 等方法都是幂等的，而 POST 方法不是。

GET /pageX HTTP/1.1 是幂等的，连续调用多次，客户端接收到的结果都是一样的：

```
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
```

POST /add\_row HTTP/1.1 不是幂等的，如果调用多次，就会增加多行记录：

```
POST /add_row HTTP/1.1    -> Adds a 1nd row
POST /add_row HTTP/1.1    -> Adds a 2nd row
POST /add_row HTTP/1.1    -> Adds a 3rd row
```

DELETE /idx/delete HTTP/1.1 是幂等的，即使不同的请求接收到的状态码不一样：

```
DELETE /idx/delete HTTP/1.1    -> Returns 200 if idx exists
DELETE /idx/delete HTTP/1.1    -> Returns 404 as it just got deleted
DELETE /idx/delete HTTP/1.1    -> Returns 404
```

## 可缓存

如果要对响应进行缓存，需要满足以下条件：

- 请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
- 响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501。
- 响应报文的 Cache-Control 首部字段没有指定不进行缓存。

## 在浏览器中输入url地址 ->> 显示主页的过程

百度好像最喜欢问这个问题。

打开一个网页，整个过程会使用哪些协议

图解（图片来源：《图解HTTP》）：

过程	使用的协议
1. 浏览器查找域名的IP地址 (DNS查找过程: 浏览器缓存、路由器缓存、DNS 缓存)	DNS: 获取域名对应IP
2. 浏览器向web服务器发送一个HTTP请求 (cookies会随着请求发送给服务器)	
3. 服务器处理请求 (请求 处理请求 & 它的参数、cookies、生成一个HTML 响应)	<ul style="list-style-type: none"> <li>• TCP: 与服务器建立TCP连接</li> <li>• IP: 建立TCP协议时, 需要发送数据, 发送数据在网络层使用IP协议</li> <li>• OPSF: IP数据包在路由器之间, 路由选择使用OPSF协议</li> <li>• ARP: 路由器在与服务器通信时, 需要将ip地址转换为MAC地址, 需要使用ARP协议</li> <li>• HTTP: 在TCP建立完成后, 使用HTTP协议访问网页</li> </ul>
4. 服务器发回一个HTML响应	
5. 浏览器开始显示HTML	

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

具体可以参考下面这篇文章: <https://segmentfault.com/a/1190000006879700>

## 常见HTTP状态码

- 1xx: 指示信息--表示请求已接收, 继续处理
- 2xx: 成功--表示请求已被成功接收、理解、接受
  - 200 OK: 正常返回信息
- 3xx: 重定向--要完成请求必须进行更进一步操作
- 4xx: 客户端错误--请求有语法错误或请求无法实现
  - 400 Bad Request: 客户端请求有语法错误, 不能被服务器理解
  - 401 Unauthorized: 请求未经授权, 这个状态码必须和WWW-Authenticate报头域一起使用
  - 403 Forbidden: 服务器收到请求, 但拒绝提供服务
  - 404 Not Found: 请求资源不存在
- 5xx: 服务器端错误--服务器未能实现合法的请求
  - 500 Internal Server Error: 服务器发生不可预期的错误
  - 503 Server Unavailable: 服务器当前不可处理客户端请求, 一段时间后可能恢复正常

## Cookie和Session

Cookie：服务器发给客户端的特殊信息，以文本形式存放在客户端。客户端再次请求的时候，会把Cookie回发。服务器接收后，会解析Cookie生成与客户端相应的内容。

Session：服务器端的机制，在服务器上保存信息。解析客户端请求并操作session\_id，按需保存状态信息。

## 区别

- Cookie数据存放在客户端浏览器上，Session数据存放在服务器上
- Session相对于Cookie更安全
- 若考虑减轻服务器负担，应当使用Cookie

# RESTful API

要弄清楚什么是RESTful API，首先要弄清楚什么是REST。REST -- REpresentational State Transfer，英语的直译就是“表现层状态转移”。如果看这个概念，估计没几个人能明白是什么意思。那下面就让我来用一句人话解释一下什么是RESTful：**URL定位资源，用HTTP动词（GET,POST,PUT,DELETE）描述操作。**

Resource：资源，即数据。

Representational：某种表现形式，比如用JSON，XML，JPEG等；

State Transfer：状态变化。通过HTTP动词实现。

所以RESTful API就是REST风格的API。那么在什么场景下使用RESTful API呢？在当今的互联网应用的前端展示媒介很丰富。有手机、有平板电脑还有PC以及其他展示媒介。那么这些前端接收到的用户请求统一由一个后台来处理并返回给不同的前端肯定是最科学和最经济的方式，RESTful API就是一套协议来规范多种形式的前端和同一个后台的交互方式。

RESTful API由后台也就是SERVER来提供前端来调用。前端调用API向后台发起HTTP请求，后台响应请求将处理结果反馈给前端。也就是说RESTful是典型的基于HTTP的协议。那么RESTful API有哪些设计原则和规范呢？

- 资源。首先是弄清楚资源的概念。资源就是网络上的一个实体，一段文本，一张图片或者一首歌曲。资源总是要通过一种载体来反应它的内容。文本可以用TXT，也可以用HTML或者XML、图片可以用JPG格式或者PNG格式，JSON是现在最常用的资源表现形式。
- 统一接口。RESTful风格的数据元操CRUD（create,read,update,delete）分别对应HTTP方法：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源，这样就统一了数据操作的接口。
- URI。可以用一个URI（统一资源定位符）指向资源，即每个URI都对应一个特定的资源。要获取这个资源访问它的URI就可以，因此URI就成了每一个资源的地址或识别符。一般的，每个资源至少有一个URI与之对应，最典型的URI就是URL。
- 无状态。所谓无状态即所有的资源都可以URI定位，而且这个定位与其他资源无关，也不会因为其他资源的变化而变化。有状态和无状态的区别，举个例子说明一下，例如要查询员工工资的步骤为第一步：登录系统。第二步：进入查询工资的页面。第三步：搜索该员工。第四步：点击姓名查看工资。这样的操作流程就是有状态的，查询工资的每一个步骤都依赖于前一个步骤，只要前置操作不成功，后续操作就无法执行。如果输入一个URL就可以得到指定员工的工资，则这种情况就是无状态的，因为获取工资不依赖于其他资源或状态，且这种情况下，员工工资是一个资源，由一个URL与之对应可以通过HTTP中的GET方法得到资源，这就是典型的RESTful风格。

说了这么多，到底RESTful长什么样子的呢？

GET：<http://www.xxx.com/source/id> 获取指定ID的某一类资源。例如GET：<http://www.xxx.com/friends/123> 表示获取ID为123的会员的好友列表。如果不加id就表示获取所有会员的好友列表。

POST：<http://www.xxx.com/friends/123> 表示为指定ID为123的会员新增好友。其他的操作类似就不举例了。

RESTful API还有其他一些规范。1：应该将API的版本号放入URL。GET:<http://www.xxx.com/v1/friend/123>。或者将版本号放在HTTP头信息中。我个人觉得要不要版本号取决于自己开发团队的习惯和业务的需要，不是强制的。2：URL中只能有名词而不能有动词，操作的表达是使用HTTP的动词GET,POST,PUT,DELETE。URL只标识资源的地址，既然是资源那就是名词了。3：如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。`?limit=10`：指定返回记录的数量、`?page=2&per_page=100`：指定第几页，以及每页的记录数。

优点：

1. 前后端分离，减少流量
2. 安全问题集中在接口上，由于接受json格式，防止了注入型等安全问题
3. 前端无关化，后端只负责数据处理，前端表现方式可以是任何前端语言（android, ios, html5）
4. 前端和后端人员更加专注于各自开发，只需接口文档便可完成前后端交互，无需过多相互了解
5. 服务器性能优化：由于前端是静态页面，通过nginx便可获取，服务器主要压力放在了接口上

## HTTP和RPC

---

在HTTP和RPC的选择上，可能有些人是迷惑的，主要是因为，有些RPC框架配置复杂，如果走HTTP也能完成同样的功能，那么为什么要选择RPC，而不是更容易上手的HTTP来实现了。

本文主要来阐述HTTP和RPC的异同，让大家更容易根据自己的实际情况选择更适合的方案。

- 传输协议
  - RPC，可以基于TCP协议，也可以基于HTTP协议
  - HTTP，基于HTTP协议
- 传输效率
  - RPC，使用自定义的TCP协议，可以让请求报文体积更小，或者使用HTTP2协议，也可以很好的减少报文的体积，提高传输效率
  - HTTP，如果是基于HTTP1.1的协议，请求中会包含很多无用的内容，如果是基于HTTP2.0，那么简单的封装以下是可以作为一个RPC来使用的，这时标准RPC框架更多的是服务治理
- 性能消耗，主要在于序列化和反序列化的耗时
  - RPC，可以基于thrift实现高效的二进制传输
  - HTTP，大部分是通过json来实现的，字节大小和序列化耗时都比thrift要更消耗性能
- 负载均衡
  - RPC，基本都自带了负载均衡策略
  - HTTP，需要配置Nginx, HAProxy来实现
- 服务治理（下游服务新增，重启，下线时如何不影响上游调用者）
  - RPC，能做到自动通知，不影响上游
  - HTTP，需要事先通知，修改Nginx/HAProxy配置

总结：

RPC主要用于公司内部的服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器接口调用，APP接口调用，第三方接口调用等。

# 计算机网络

## 计算机网络体系结构



 CyC2018

## 五层协议

- 应用层：为特定应用程序提供数据传输服务，例如 HTTP、DNS 等协议。数据单位为报文。
- 运输层：为进程提供通用的数据传输服务，提供端对端的接口。传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为报文段；用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。
- 网络层：为主机提供数据传输服务，为数据包选择路由。传输单位为数据包，路由器工作在这一层。
- 数据链路层：网络层针对的还是主机之间的数据传输服务，而主机之间可以有很多链路，链路层协议就是为同一链路的主机提供数据传输服务。格式化数据，控制对物理介质的访问，错误检测/纠正（保证传输可靠性）。传输单位为帧，交换机工作在这一层。
- 物理层：考虑的是怎样在传输媒体上上传输数据比特流，而不是指具体的传输媒体。

## OSI

其中表示层和会话层用途如下：

- **表示层**：数据压缩、加密以及数据描述，这使得应用程序不必关心在各台主机中数据内部格式不同的问题。
- **会话层**：建立及管理会话。

五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

OSI仅仅是一种理论规范（通信协议必要的功能是什么），一种概念型框架，实际开发中用到的是TCP/IP（在计算机上实现协议应该开发哪种程序）

## TCP 的三次握手

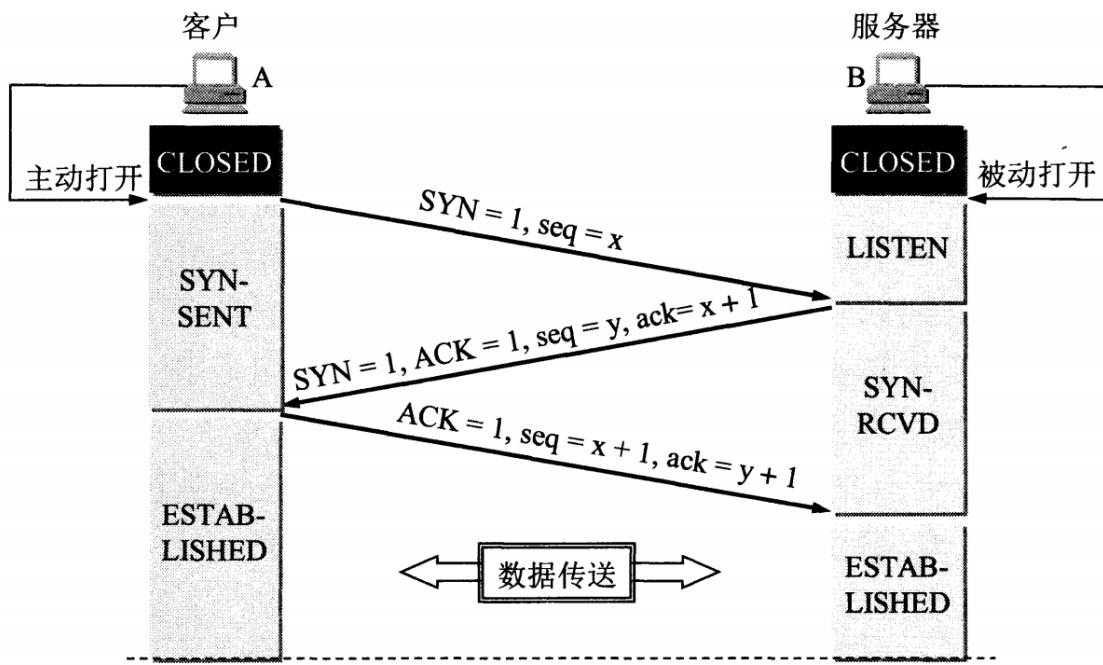


图 5-28 用三报文握手建立 TCP 连接

假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文， $SYN=1$ ,  $ACK=0$ ，选择一个初始的序号  $x$ 。
- B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文， $SYN=1$ ,  $ACK=1$ ，确认号为  $x+1$ ，同时也选择一个初始的序号  $y$ 。
- A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为  $y+1$ ，序号为  $x+1$ 。
- B 收到 A 的确认后，连接建立。

### 三次握手的原因

三次握手才能确认双方的接受与发送能力是否正常，而两次却不可以。第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。第二次握手：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

为了初始化Sequence Number的初始值。作为以后数据通信的序号，保证应用层接收到的数据不会因网络的原因而乱序，TCP会用这个序号拼接数据。

如果是 HTTPS 协议的话，三次握手这个过程，还会进行数字证书的验证以及加密密钥的生成。

## 三次握手过程中可以携带数据吗

很多人可能会认为三次握手都不能携带数据，其实第三次握手的时候，是可以携带数据的。

也就是说，第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。

为什么这样呢？大家可以想一个问题，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据。

因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。

也就是说，第一次握手可以放数据的话，其中一个简单的原因就是会让服务器更加容易受到攻击了。

而对于第三次的话，此时客户端已经处于 established 状态，也就是说，对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据页没啥毛病。

## 首次握手的隐患——SYN超时

原因分析

- Server收到Client的SYN，回复SYN-ACK的时候未收到ACK确认
- Server不断重试直到超时，Linux默认等待63秒才断开连接

可能会使Server受到SYN Flood的风险，针对其防护措施

- 超时时间缩短
- 连续收到连续某个IP的重复SYN报文，判断受到攻击，丢弃该IP发送的包

## 建立连接后，Client出现故障

保活机制

- 向对方发送保活探测报文，如未收到响应则继续发送
- 尝试次数达到保活探测数仍未收到相应则中断连接

## 什么是半连接队列

服务器第一次收到客户端的 SYN 之后，就会处于 SYN\_RECV 状态，此时双方还没有完全建立其连接，服务器会把此种状态下请求连接放在一个队列里，我们把这种队列称之为半连接队列。

当然还有一个全连接队列，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

# TCP四次挥手

---

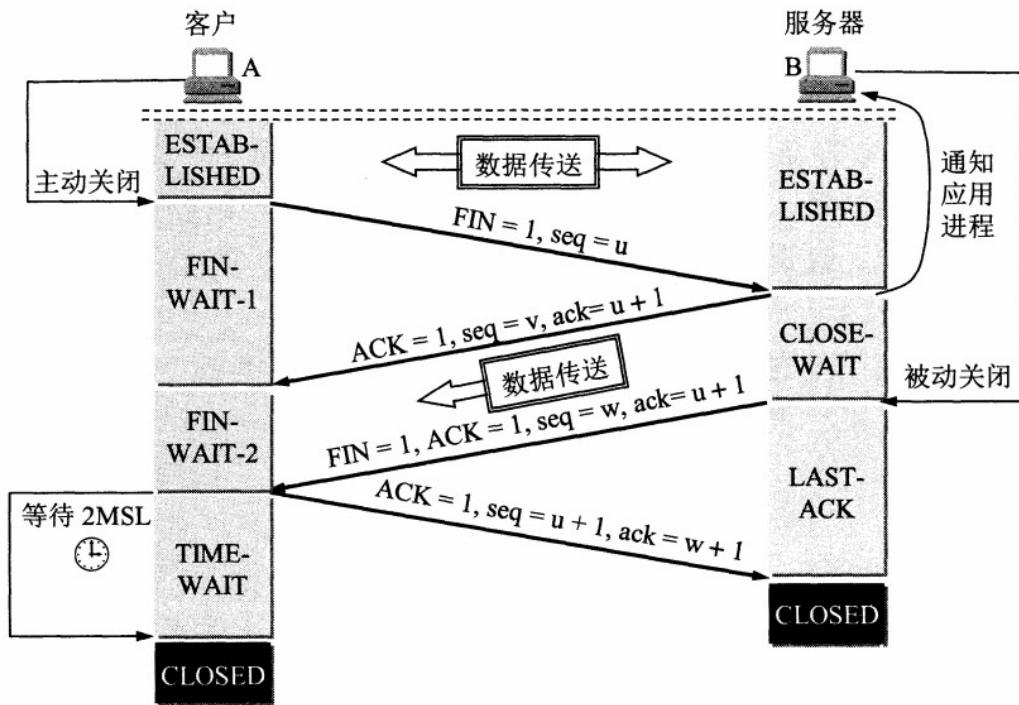


图 5-29 TCP 连接释放的过程

以下描述不讨论序号和确认号，因为序号和确认号的规则比较简单。并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

- A 发送连接释放报文,  $FIN=1$ 。
- B 收到之后发出确认, 此时 TCP 属于半关闭状态, B 能向 A 发送数据但是 A 不能向 B 发送数据。
- 当 B 不再需要连接时, 发送连接释放报文,  $FIN=1$ 。
- A 收到后发出确认, 进入 TIME-WAIT 状态, 等待 2 MSL (最大报文存活时间) 后释放连接。
- B 收到 A 的确认后释放连接。

#### 四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

#### TIME\_WAIT

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

#### 服务器出现大量CLOSE\_WAIT状态的原因

- 对方关闭socket连接，我方忙于读或写，没有及时关闭连接
  - 检查代码，特别是释放资源的代码
  - 检查配置，特别是处理请求的线程配置

## TCP 协议如何保证可靠传输

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

3. **校验和**: TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. **流量控制**: TCP 连接的每一方都有固定大小的缓冲空间，TCP 的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）
6. **拥塞控制**: 当网络拥塞时，减少数据的发送。
7. **ARQ协议**: 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. **超时重传**: 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

## TCP, UDP 协议的区别

类型	特点			性能		应用场景	首部字节
	是否面向连接	传输可靠性	传输形式	传输效率	所需资源		
TCP	面向连接	可靠	字节流	慢	多	要求通信数据可靠 (如文件传输、邮件传输)	20-60
UDP	无连接	不可靠	数据报文段	快	少	要求通信速度高 (如域名转换)	8个字节 (由4个字段组成)

### TCP

- 面向连接的、可靠的、基于字节流的传输层通信协议
- 将应用层的数据流分割成报文段并发给目标节点的TCP层
- 数据包都有序号，对方收到则发送ACK确认，未收到则重传
- 使用校验和来检验数据在传输过程中是否有误

### UDP

- 面向非连接
- 不维护连接状态，支持同时向多个客户端传输相同的消息
- 数据包报头只有8字节，额外开销较小
- 吞吐量只受限于数据生成速率、传输速率以及机器性能
- 尽最大努力交付，不保证可靠交付，不需要维持复杂的链接状态表

## TCP 滑动窗口

- RTT: 发送一个数据包到收到对应的ACK所花费的时间
- RTO: 重传时间间隔

滑动窗口协议是传输层进行流量控制和乱序重排的一种措施，接收方通过通告发送方自己的窗口大小，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。

ACK包含两个非常重要的信息：一是期望接收到的下一字节的序号n，该n代表接收方已经接收到了前n-1字节数据。二是当前的窗口大小m，如此发送方在接收到ACK包含的这两个数据后就可以计算出还可以发送多少字节的数据给对方，这就是滑动窗口控制流量的基本原理

TCP的滑动窗口是动态的，应用程序在需要（如内存不足）时，通过API通知TCP协议栈缩小TCP的接收窗口。然后TCP协议栈在下个段发送时包含新的窗口大小通知给对端，对端按通知的窗口来改变发送窗口，以此达到减缓发送速率的目的。

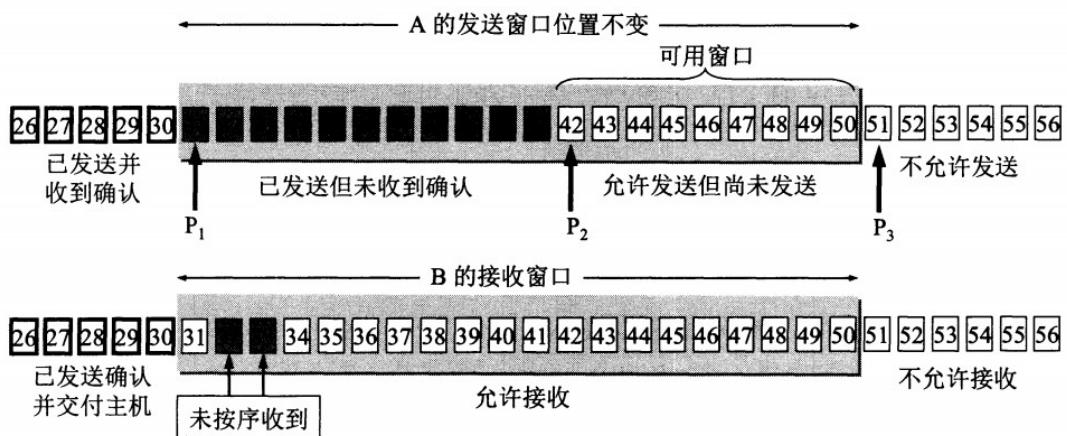
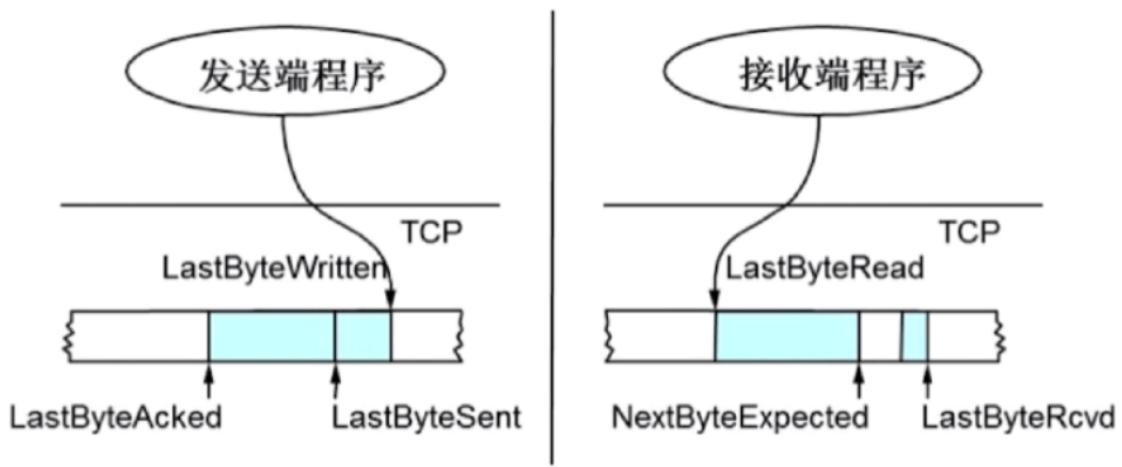


图 5-16 A 发送了 11 个字节的数据



- 接收端还能处理的数据量  $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$
- 窗口内剩余可发送数据量  $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$

## TCP 流量控制

流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

## TCP 拥塞控制

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

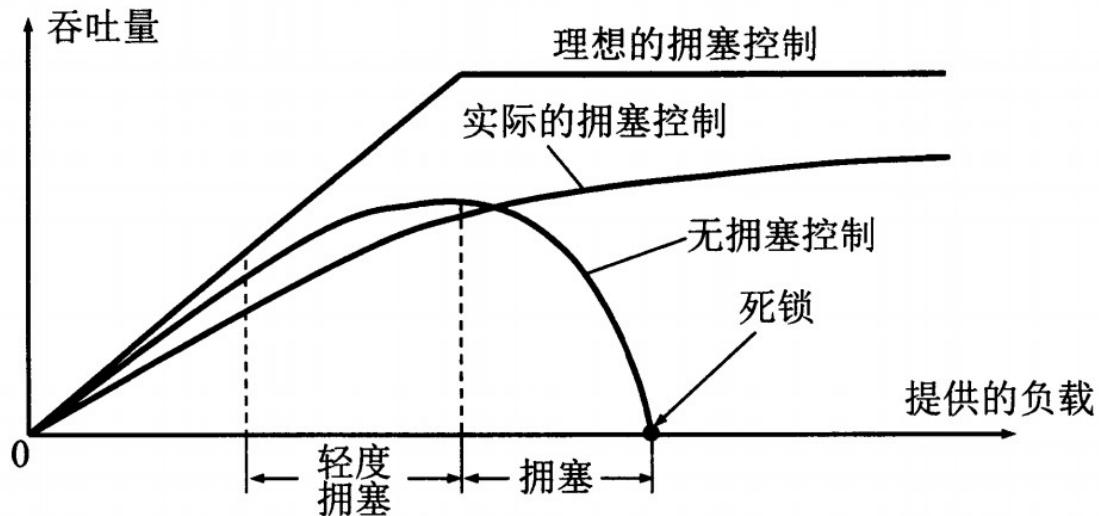


图 5-23 拥塞控制所起的作用

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口 (cwnd) 的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

- 接收方有足够的接收缓存，因此不会发生流量控制；
- 虽然 TCP 的窗口基于字节，但是这里设窗口的大小单位为报文段。

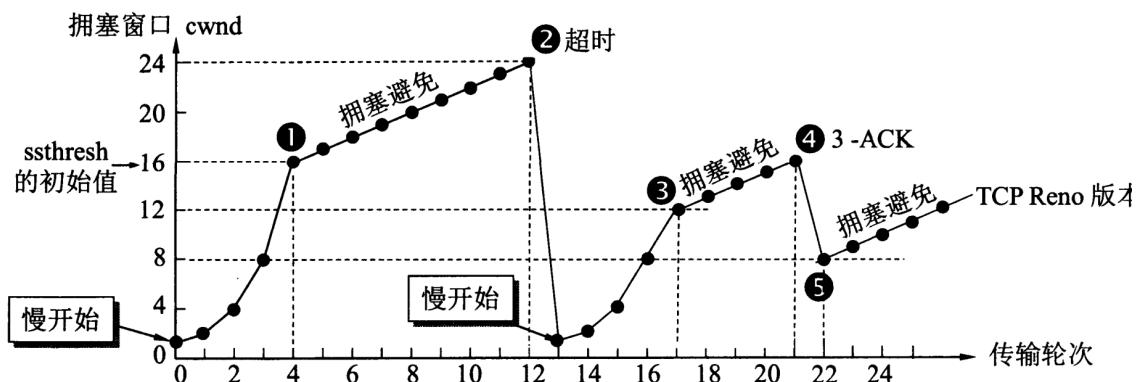


图 5-25 TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

## 1. 慢开始与拥塞避免

发送的最初执行慢开始，令  $cwnd = 1$ ，发送方只能发送 1 个报文段；当收到确认后，将  $cwnd$  加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将  $cwnd$  加倍，这样会让  $cwnd$  增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限  $ssthresh$ ，当  $cwnd \geq ssthresh$  时，进入拥塞避免，每个轮次只将  $cwnd$  加 1。

如果出现了超时，则令  $ssthresh = cwnd / 2$ ，然后重新执行慢开始。

## 2. 快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M<sub>2</sub>，则 M<sub>3</sub> 丢失，立即重传 M<sub>3</sub>。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2，cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

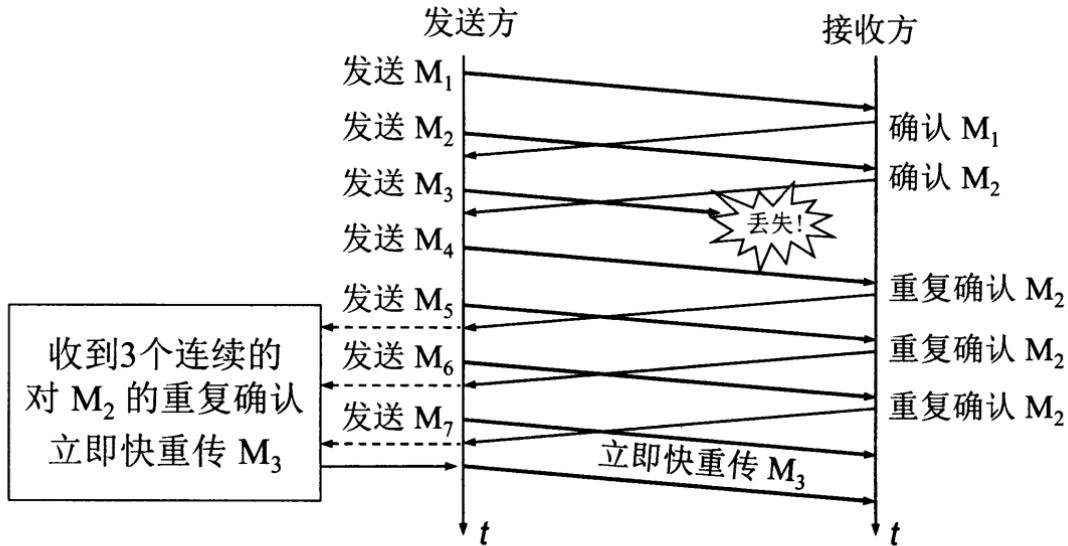


图 5-26 快重传的示意图

## 交换机和路由器

主要的区别体现在以下几个方面：

(1) 外形上：

从外形上我们区分两者，交换机通常端口比较多看起来比较笨重，而路由器的端口就少得多体积也小得多，实际上右图并不是真正的路由器只是集成了路由器的功能，除此之外还有交换机的功能（LAN口就是作为交换机的端口来使用，WAN是用于连接外网的端口），而两个天线则是无线AP接入点（即是通常所说的无线局域网wifi）。

(2) 工作层次不同：

最初的交换机工作在OSI开放式系统互联模型的数据链路层，也就是第二层，而路由器则工作在OSI模型的网络层，就是第三层。也就是由于这一点所以交换机的原理比较简单，一般都是采用硬件电路实现数据帧的转发，而路由器工作在网络层，肩负着网络互联的重任，要实现更加复杂的协议，具有更加智能的转发决策功能，一般都会在路由器中跑操作系统，实现复杂的路由算法，更偏向于软件实现其功能。

(3) 数据的转发对象不同：

交换机是根据MAC地址转发数据帧，而路由器则是根据IP地址来转发IP数据报/分组。数据帧是在IP数据包/分组的基础上封装了帧头（源MAC和目的MAC等）和帧尾（CRC校验码）。而对于MAC地址和IP地址大家也许就搞不明白了，为何需要两个地址，实际上IP地址决定最终数据包要到达某一台主机，而MAC地址则是决定下一跳将要交互给哪一台设备（一般是路由器或主机）。而且，IP地址是软件实现的，可以描述主机所在的网络，MAC地址是硬件实现的，每一个网卡在出厂的时候都会将全世界唯一的MAC地址固化在网卡的ROM中，所以MAC地址是不能被修改的，但是IP地址是可以被网络管理人员配置修改的。

#### (4) "分工"不同

交换机主要是用于组建局域网，而路由器则是负责让主机连接外网。多台主机可以通过网线连接到交换机，这时就组建好了局域网，就可以将数据发送给局域网中的其他主机，如我们使用的飞秋、极域电子教室等局域网软件就是通过交换机把数据转发给其他主机的，当然像极域电子教室这样的广播软件是利用广播技术让所有的主机都收到数据的。然而，通过交换机组建的局域网是不能访问外网的（即是Internet），这时需要路由器来为我们“打开外面精彩世界的大门”，局域网的所有主机使用的都是私网的IP，所以必须通过路由器转化为公网的IP之后才能访问外网。

#### (5) 冲突域和广播域

交换机分割冲突域，但是不分割广播域，而路由器分割广播域。由交换机连接的网段仍属于同一个广播域，广播数据包会在交换机连接的所有网段上传播，在这种情况下会导致广播风暴和安全漏洞问题。而连接在路由器上的网段会被分配不通的广播域，路由器不会转发广播数据。需要说明的是单播的数据包在局域网中会被交换机唯一地送往目标主机，其他主机不会接收到数据，这是区别于原始的集线器的，数据的到达时间由交换机的转发速率决定，交换机会转发广播数据给局域网中的所有主机。

最后需要说明的是：路由器一般有防火墙的功能，能够对一些网络数据包选择性过滤。现在的一些路由器都具备交换机的功能（如上图右），一些交换机具备路由器的功能，被称为3层交换机，广泛使用。相比较而言，路由器的功能较交换机要强大，但是速度也较慢，价格昂贵，三层交换机既有交换机的线性转发报文的能力，又有路由器的良好的路由功能因此得到广泛的使用。

当然关于路由器和交换机的一些介绍远不止这些，上述所说是主要的一些区别，同时也是本人对路由器和交换机的浅显认识，如有其他一些较明显的区别特征望给出宝贵意见。

## DNS

### DNS占用53号端口，同时使用TCP和UDP协议。那么DNS在什么情况下使用这两种协议？

DNS在区域传输的时候使用TCP协议，其他时候使用UDP协议。

DNS区域传输的时候使用TCP协议：

辅域名服务器会定时（一般3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，会执行一次区域传送，进行数据同步。区域传送使用TCP而不是UDP，因为数据同步传送的数据量比一个请求应答的数据量要多得多。TCP是一种可靠连接，保证了数据的准确性。

域名解析时使用UDP协议：

客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过三次握手，这样DNS服务器负载更低，响应更快。理论上说，客户端也可以指定向DNS服务器查询时用TCP，但事实上，很多DNS服务器进行配置的时候，仅支持UDP查询包。

### DNS查询

域名服务器分类	作用
根域名服务器	最高层次的域名服务器，本地域名服务器解析不了的域名就会向其求助
顶级域名服务器	负责管理在该顶级域名服务器下注册的二级域名
权限域名服务器	负责一个区的域名解析工作
本地域名服务器	当一个主机发出DNS查询请求时，这个查询请求首先发给本地域名服务器

(1) 递归查询：本机向本地域名服务器发出一次查询请求，就静待最终的结果。如果本地域名服务器无法解析，自己会以DNS客户机的身份向其它域名服务器查询，直到得到最终的IP地址告诉本机。

(2) 迭代查询：本地域名服务器向根域名服务器查询，根域名服务器告诉它下一步到哪里去查询，然后它再去查，每次它都是以客户机的身份去各个服务器查询。



# 10亿级别订单的分库分表方案

## 背景

随着公司业务增长，如果每天1000多万笔订单的话，3个月将有约10亿的订单量，之前数据库采用单库单表的形式已经不满足于业务需求，数据库改造迫在眉睫。

## 订单数据如何划分？

我们可以将订单数据划分成两大类型：分别是热数据和冷数据。

热数据：3个月内的订单数据，查询实时性较高。

冷数据A：3个月 ~ 12个月前的订单数据，查询频率不高。

冷数据B：1年前的订单数据，几乎不会查询，只有偶尔的查询需求。

可能这里有个疑惑为什么要将冷数据分成两类，因为根据实际场景需求，用户基本不会去查看1年前的数据，如果将这部分数据还存储在db中，那么成本会非常高，而且也不便于维护。另外如果真遇到有个别用户需要查看1年前的订单信息，可以让用户走离线数据查看。

对于这三类数据的存储，目前规划如下：

热数据：使用mysql进行存储，当然需要分库分表

冷数据A：对于这类数据可以存储在ES中，利用搜索引擎的特性基本上也可以做到比较快的查询。

冷数据B：对于这类不经常查询的数据，可以存放到hive中

## MySql 如何分库分表

### 一、按业务拆分

在业务初始阶段，为了加快应用上线和快速迭代，很多应用都采用集中式的架构。但是随着业务系统的扩大，系统匾额越来越复杂，越来越难以维护，开发效率变得越来越低，并且对资源的消耗也变得越来越大，通过硬件提高系统性能的成本会变得更高。

通常一般的电商平台，包含了用户、商品、订单等几大模块，简单的做法是在同一个库中分别建4张表。

但是随着业务的提升，将所有业务都放在一个库中已经变得越来越难以维护，因此我们建议，将不同业务放在不同的库中。

我们将不同的业务放到不同的库中，将原来所有压力由同一个库中分散到不同的库中，提升了系统的吞吐量。

### 二、分库与分表

我们知道每台机器无论配置多么好它都有自身的物理上限，所以当我们应用已经能触及或远远超出单台机器的某个上限的时候，我们惟有寻找别的机器的帮助或者继续升级的我们的硬件，但常见的方案还是通过添加更多的机器来共同承担压力。

我们还得考虑当我们的业务逻辑不断增长，我们的机器能不能通过线性增长就能满足需求？因此，使用数据库的分库分表，能够立竿见影的提升系统的性能，关于为什么要使用数据库的分库分表的其他原因这里不再赘述，主要讲具体的实现策略。

## 1)、分表策略

我们以订单表为例，在订单表中，订单id肯定是不可重复的，因此将该字段当做shard key 是非常适合的，其他表类似。假设订单表的字段如下：

```
create table order(
    order_id bigint(11) ,
    user_id bigint(11),
    phone varchar(15),
    ...
)
```

我们假设预估单个库需要分配100个表满足我们的业务需求，我们可以简单的取模计算出订单在哪个子表中，例如：`order_id % 100`。

这时候可能会有人问了，如果我根据`order_id`进行分表规则，但是我想根据`user_id`查询相应的订单，不是定位不到哪个子表了吗，的确是这样，一旦确定shard key，就只能根据shard key定位到子表进而查询该子表下的数据；如果确实想根据`user_id`去查询相关订单，那应该将shard key设置为`user_id`，那分表规则也相应的变更为：`user_id % 100`；

## 2)、分库实现策略

数据库分表能够解决单表数据量很大的时候数据查询的效率问题，但是无法给数据库的并发操作带来效率上的提高，因为分表的实质还是在一个数据库上进行的操作，很容易受数据库IO性能的限制。

因此，如何将数据库IO性能的问题平均分配出来，很显然将数据进行分库操作可以很好地解决单台数据库的性能问题。

分库策略与分表策略的实现很相似，最简单的都是可以通过取模的方式进行路由。

我们还是以`order`表举例，

例如：`order_id % 库容量`,

如果`order_id`不是整数类型，可以先hash在进行取模，

例如：`hash(order_id) % 库容量`

## 3)、分库分表结合使用策略

数据库分表可以解决单表海量数据的查询性能问题，分库可以解决单台数据库的并发访问压力问题。有时候，我们需要同时考虑这两个问题，因此，我们既需要对单表进行分表操作，还需要进行分库操作，以便同时扩展系统的并发处理能力和提升单表的查询性能，就是我们使用到的分库分表。

如果使用分库分表结合使用的话，不能简单进行`order_id`取模操作，需要加一个中间变量用来打散到不同的子表，公式如下：

1中间变量 = shard key % (库数量\*单个库的表数量)；

2库序号 = 取整 (中间变量 / 单个库的表数量)；

3表序号 = 中间变量%单个库的表数量；

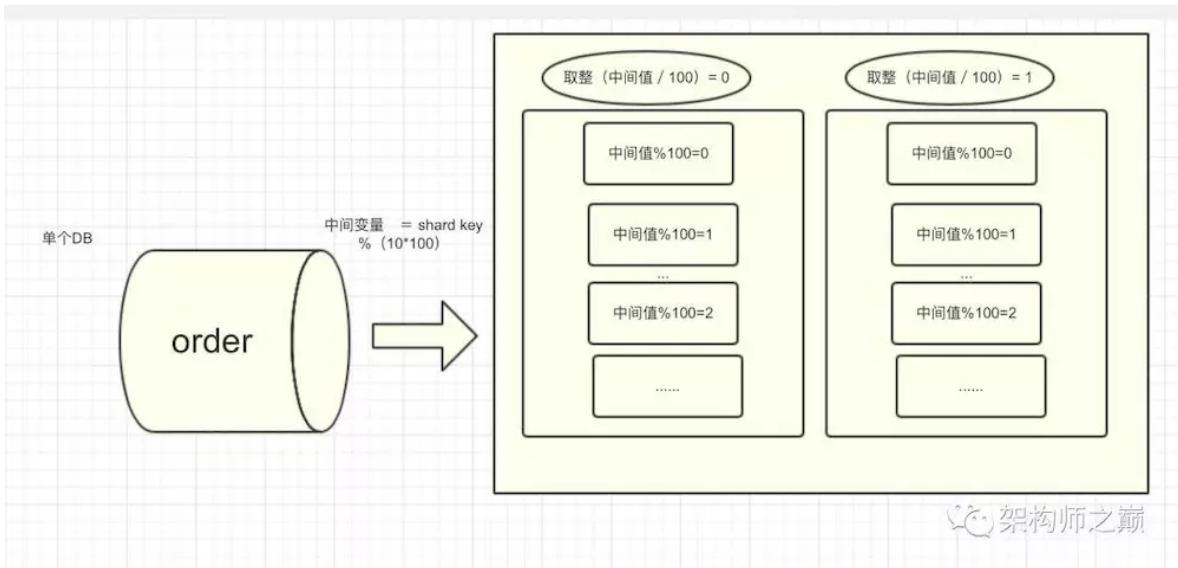
例如：数据库有10个，每一个库中有100个数据表，用户的`order_id = 1001`，按照上述的路由策略，可得：

1中间变量 =  $1001 \% (10 \times 100) = 1$ ;

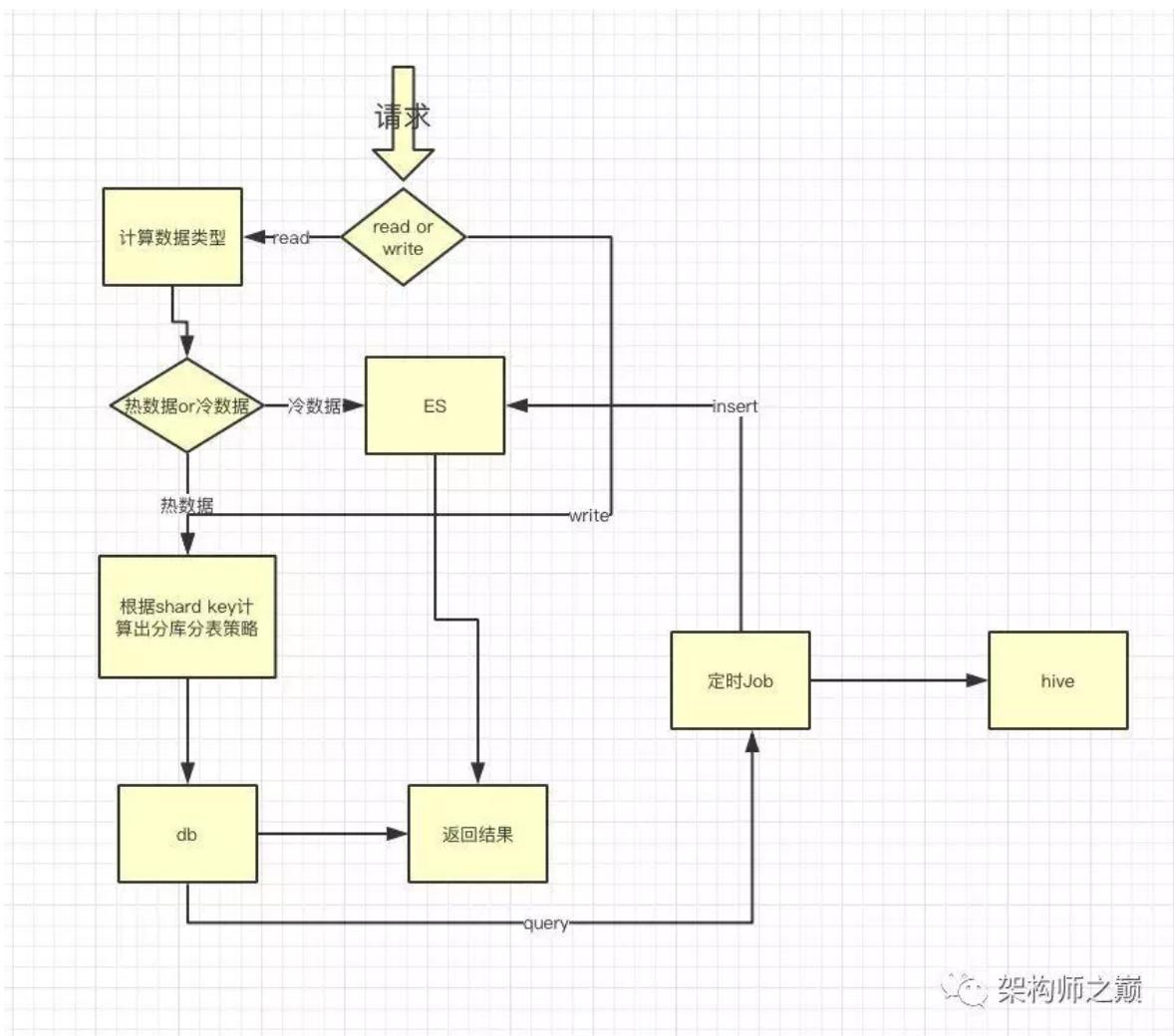
2库序号 = 取整 ( $1 / 100$ ) = 0;

3表序号 =  $1 \% 100 = 1$ ;

这样的话，对于order\_id = 1001，将被路由到第1个数据库的第2个表中(索引0代表1，依次类推)。



### 整体架构设计



从图中我们将请求分成read和write请求，write请求比较简单，就是根据分库分表规则写入db即可。

对于read请求，我们需要计算出查询的是热数据还是冷数据，一般order\_id生成规则如下，“商户所在地区号+时间戳+随机数”，我们可以根据时间戳计算出查询的是热数据还是冷数据，（当然具体业务需要具体对待，这里不再详细阐述）

另外架构图中的冷数据指的是3个月~12个月前的数据，如果是查询一年前的数据，建议直接离线查hive即可。

图中有一个定时Job，主要用来定时迁移订单数据，需要将冷数据分别迁移到ES和hive中。

# Redis 如何保持和 MySQL 数据一致

## 1. MySQL持久化数据，Redis只读数据

redis在启动之后，从数据库加载数据。

**读请求：**

不要求强一致性的读请求，走redis，要求强一致性的直接从mysql读取

**写请求：**

数据首先都写到数据库，之后更新redis（先写redis再写mysql，如果写入失败事务回滚会造成redis中存在脏数据）

## 2. MySQL和Redis处理不同的数据类型

- MySQL处理实时性数据，例如金融数据、交易数据
- Redis处理实时性要求不高的数据，例如网站最热贴排行榜，好友列表等

在并发不高的情况下，读操作优先读取redis，不存在的话就去访问MySQL，并把读到的数据写回Redis中；写操作的话，直接写MySQL，成功后再写入Redis(可以在MySQL端定义CRUD触发器，在触发CRUD操作后写数据到Redis，也可以在Redis端解析binlog，再做相应的操作)

在并发高的情况下，读操作和上面一样，写操作是异步写，写入Redis后直接返回，然后定期写入MySQL

**几个例子：**

**当更新数据时，如更新某商品的库存，当前商品的库存是100，现在要更新为99，先更新数据库更改成99，然后删除缓存，发现删除缓存失败了，这意味着数据库库存的是99，而缓存是100，这导致数据库和缓存不一致。**

**解决方法：**

这种情况应该是先删除缓存，然后在更新数据库，如果删除缓存失败，那就不要更新数据库，如果说删除缓存成功，而更新数据库失败，那查询的时候只是从数据库里查了旧的数据而已，这样就能保持数据库与缓存的一致性。

**为什么是删除缓存，而不是更新缓存？**

原因很简单，很多时候，在复杂的缓存场景，缓存不单单是数据库中直接取出来的值。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

另外更新缓存的代价有时候是很高的。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于**比较复杂的缓存数据计算的场景**，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，**这个缓存到底会不会被频繁访问到？**

举个栗子，一个缓存涉及的表的字段，在1分钟内就修改了20次，或者是100次，那么缓存更新20次、100次；但是这个缓存在1分钟内只被读取了1次，有大量的冷数据。实际上，如果你只是删除缓存的话，那么在1分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。**用到缓存才去算缓存。**

其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊。80%的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工。

**在高并发的情况下，如果当删除完缓存的时候，这时去更新数据库，但还没有更新完，另外一个请求来查询数据，发现缓存里没有，就去数据库里查，还是以上面商品库存为例，如果数据库中产品的库存是100，那么查询到的库存是100，然后插入缓存，插入完缓存后，原来那个更新数据库的线程把数据库更新为了99，导致数据库与缓存不一致的情况**

解决方法：

遇到这种情况，可以用队列的去解决这个问题，创建几个队列，如20个，根据商品的ID去做hash值，然后对队列个数取模，当有数据更新请求时，先把它丢到队列里去，当更新完后在从队列里去除，如果在更新的过程中，遇到以上场景，先去缓存里看下有没有数据，如果没有，可以先去队列里看是否有相同商品ID在做更新，如果有也把查询的请求发送到队列里去，然后同步等待缓存更新完成。

这里有一个优化点，如果发现队列里有一个查询请求了，那么就不要放新的查询操作进去了，用一个while(true)循环去查询缓存，循环个200MS左右，如果缓存里还没有则直接取数据库的旧数据，一般情况下是可以取到的。

**在数据还未同步到「从库」的时候，由于cache miss去「从库」取到了未同步前的旧值**

当「从库」完成同步的时候再额外做一次delete cache或者set cache的操作。

如此，虽说也没有100%解决短暂的数据不一致问题，但是已经将脏数据所存在的时长降到了最低（最终由主从同步的耗时决定），并且大大减少了无谓的资源消耗。

**在高并发下要注意的问题：**

### 1、读请求时长阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回。

该解决方案，最大的风险点在于说，**可能数据更新很频繁**，导致队列中积压了大量更新操作在里面，然后**读请求会发生大量的超时**，最后导致大量的请求直接走数据库。务必通过一些模拟真实的测试，看看更新数据的频率是怎样的。

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要**部署多个服务**，每个服务分摊一些数据的更新操作。如果一个内存队列里居然会挤压100个商品的库存修改操作，每隔库存修改操作要耗费10ms去完成，那么最后一个商品的读请求，可能等待 $10 * 100 = 1000\text{ms} = 1\text{s}$ 后，才能得到数据，这个时候就导致**读请求的长时阻塞**。

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会 hang 多少时间，如果读请求在 200ms 返回，如果你计算过后，哪怕是最繁忙的时候，积压 10 个更新操作，最多等待 200ms，那还可以的。

**如果一个内存队列中可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。**

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来**实际粗略测算一下**。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

## 2、请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时 hang 在服务上，看服务能不能扛的住，需要多少机器才能扛住最大的极限情况的峰值。

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大。

## 3、多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过 Nginx 服务器路由到相同的服务实例上。

比如说，对同一个商品的读写请求，全部路由到同一台机器上。可以自己去做服务间的按照某个请求参数的 hash 路由，也可以用 Nginx 的 hash 路由功能等等。可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过 nginx 服务器路由到相同的服务实例上。

## 4、热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能会造成某台机器的压力过大。就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以其实要根据业务系统去看，如果更新频率不是太高的话，这个问题的影响并不是特别大，但是的确可能某些机器的负载会高一些。

# Spring

## IoC

IoC (Inverse of Control:控制反转) 是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由Spring框架来管理。IoC 在其他语言中也有应用，并非 Spring 特有。IoC 容器是 Spring 用来实现 IoC 的载体，IoC 容器实际上就是个 Map (key, value)，Map 中存放的是各种对象。

将对象之间的相互依赖关系交给 IOC 容器来管理，并由 IOC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。IOC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

推荐阅读：<https://www.zhihu.com/question/23277575/answer/169698662>

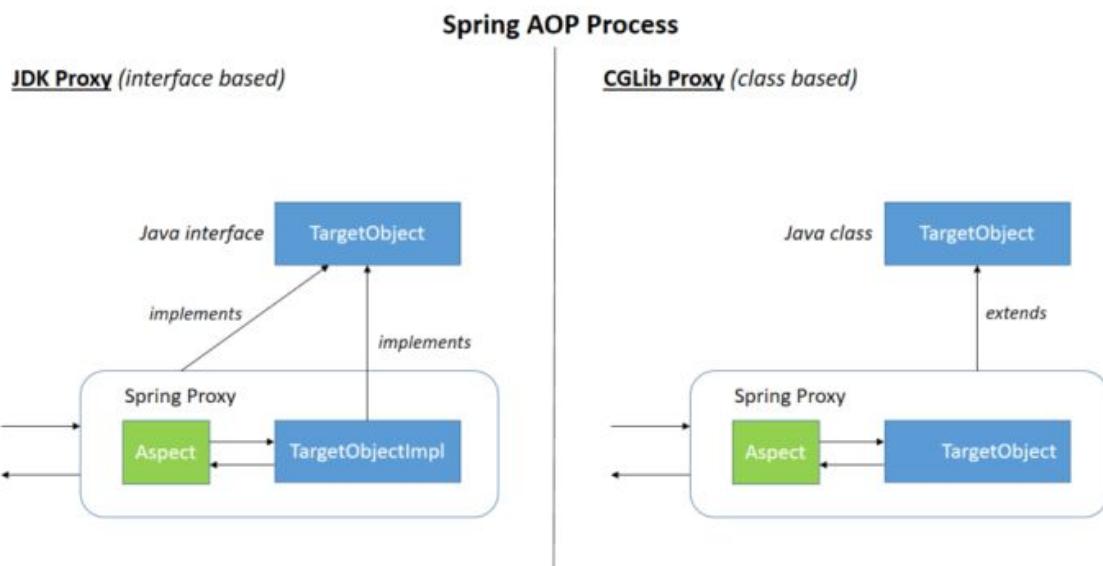
**Spring IOC的初始化过程：**



## AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用JDK Proxy去进行代理了，这时候Spring AOP会使用Cglib，这时候Spring AOP会使用 Cglib 生成一个被代理对象的子类来作为代理，如下图所示：



当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ , AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

## Bean的生命周期

---

1. Spring对Bean进行实例化
2. Spring将值和Bean的引用注入进Bean对应的属性中
3. 容器通过Aware接口把容器信息注入Bean
4. BeanPostProcessor。进行进一步的构造，会在InitializationBean前后执行对应方法，当前正在初始化的bean对象会被传递进来，我们就可以对这个bean作任何处理
5. InitializingBean。这一阶段也可以在bean正式构造完成前增加我们自定义的逻辑，但它与前置处理不同，由于该函数并不会把当前bean对象传进来，因此在这一步没办法处理对象本身，只能增加一些额外的逻辑。
6. DisposableBean。Bean将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁，如果Bean实现了接口，Spring将调用它的destory方法

# 超卖问题

---

1.不同用户在读请求的时候，发现商品库存足够，然后同时发起请求，进行秒杀操作，减库存，导致库存减为负数。

2.同一个用户在有库存的时候，连续发出多个请求，两个请求同时存在，于是生成多个订单。

对于第一种超卖现象：

(1)最简单的方法，更新数据库减库存的时候，进行库存限制条件，

```
update miaosha_goods set stock_count = stock_count - 1 where goods_id = #{goodsId} and  
stock_count > 0
```

可以简单的解决超卖的情况，但是不能完全避免；

(2)究其深层原因，是因为数据库底层的写操作和读操作可以同时进行，虽然写操作默认带有显式锁（即对同一数据不能同时进行写操作）但是读操作默认是不带锁的，所以当用户1去修改库存的时候，用户2依然可以都到库存为1，所以出现了超卖现象。

解决方案：

可以对读操作加上显式锁（即在select ...语句最后加上for update）这样一来用户1在进行读操作时用户2就需要排队等待了

但是问题来了，如果该商品很热门并发量很高那么效率就会大大的下降，怎么解决？

解决方案：

我们可以有条件有选择的在读操作上加锁，比如可以对库存做一个判断，当库存小于一个量时开始加锁，让购买者排队，这样一来就解决了超卖现象。

(3) 应用一个队列缓存，将多线程变为单线程读写。

2.第二种现象，将userId和商品Id 加上唯一索引，可以解决这种情况。插入失败。

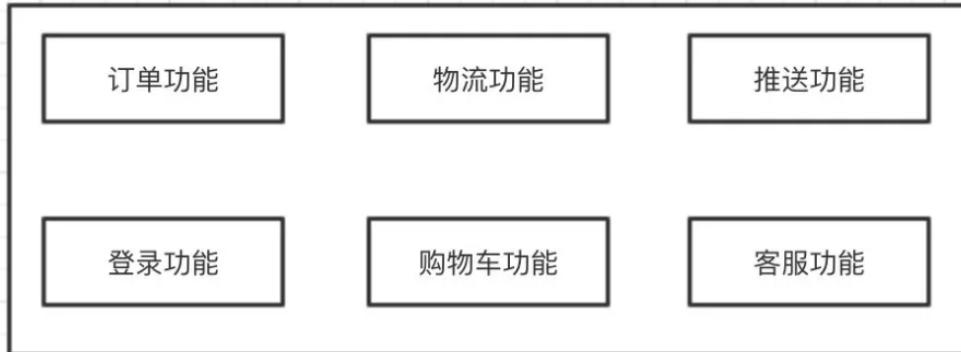
# 单点登录

## 一、什么是单点登录？

单点登录的英文名叫做：Single Sign On（简称SSO）。

在初学/以前的时候，一般我们就**单系统**，所有的功能都在同一个系统上。

所有的功能都在同一个系统上



后来，我们为了**合理利用资源和降低耦合性**，于是把单系统**拆分成**多个子系统。

- 回顾：[分布式基础知识](#)

拆分成多个子系统



比如阿里系的**淘宝**和**天猫**，很明显地我们可以知道这是两个系统，但是你在使用的时候，登录了天猫，淘宝也会自动登录。



简单来说，单点登录就是在多个系统中，用户只需一次登录，各个系统即可感知该用户已经登录。

## 二、回顾单系统登录

在我初学JavaWeb的时候，登录和注册是我做得最多的一个功能了（初学Servlet的时候做过、学SpringMVC的时候做过、跟着做项目的时候做过...），反正我也数不清我做了多少次登录和注册的功能了...这里简单讲述一下我们初学时是怎么做登录功能的。

### HTTP是无状态的协议

众所周知，HTTP是**无状态**的协议，这意味着**服务器无法确认用户的信息**。于是乎，W3C就提出了：给每一个用户都发一个通行证，无论谁访问的时候都需要携带通行证，这样服务器就可以从通行证上确认用户的信息。通行证就是**Cookie**。

如果说Cookie是检查用户身上的“通行证”来确认用户的身份，那么Session就是通过检查服务器上的“客户明细表”来确认用户的身份的。**Session相当于在服务器中建立了一份“客户明细表”**。

HTTP协议是无状态的，Session不能依据HTTP连接来判断是否为同一个用户。于是乎：服务器向用户浏览器发送了一个名为JSESSIONID的Cookie，它的值是Session的id值。**其实Session是依据Cookie来识别是否是同一个用户**。

所以，一般我们单系统实现登录会这样做：

- **登录**：将用户信息保存在Session对象中
- - 如果在Session对象中能查到，说明已经登录
- 如果在Session对象中查不到，说明没登录（或者已经退出了登录）
- **注销（退出登录）**：从Session中删除用户的信息
- **记住我（关闭掉浏览器后，重新打开浏览器还能保持登录状态）**：配合Cookie来用

我之前Demo的代码，可以参考一下：

```
/**
 * 用户登陆
 */
@PostMapping(value = "/user/session", produces = {"application/json;charset=UTF-8"})
public Result login(String mobileNo, String password, String inputCaptcha,
HttpSession session, HttpServletResponse response) {
```

```
//判断验证码是否正确
if (webutils.validateCaptcha(inputCaptcha, "captcha", session)) {

    //判断有没有该用户
    User user = userService.userLogin(mobileNo, password);
    if (user != null) {
        /*设置自动登陆，一个星期。将token保存在数据库中*/
        String loginToken = webutils.md5(new Date().toString() +
session.getId());
        user.setLoginToken(loginToken);
        User user1 = userService.userUpload(user);

        session.setAttribute("user", user1);

        CookieUtil.addCookie(response, "loginToken", loginToken, 604800);

        return ResultUtil.success(user1);

    } else {
        return ResultUtil.error(ResultEnum.LOGIN_ERROR);
    }
} else {
    return ResultUtil.error(ResultEnum.CAPTCHA_ERROR);
}

}

/***
 * 用户退出
 */
@DeleteMapping(value = "/session", produces = {"application/json;charset=UTF-
8"})
public Result logout(HttpServletRequest session,HttpServletResponse
request,HttpServletResponse response ) {

    //删除session和cookie
    session.removeAttribute("user");

    CookieUtil.clearCookie(request, response, "loginToken");

    return ResultUtil.success();
}
/***
 * @author ozc
 * @version 1.0
 * <p>
 * 拦截器：实现自动登陆功能
 */
public class UserInterceptor implements HandlerInterceptor {

    @Autowired
    private UserService userService;

    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object o) throws Exception {
        User sessionUser = (User) request.getSession().getAttribute("user");
    }
}
```

```

// 已经登陆了，放行
if (sessionUser != null) {
    return true;
} else {
    //得到带过来cookie是否存在
    String loginToken = Cookieutil.findCookieByName(request, "loginToken");
    if (StringUtils.isNotBlank(loginToken)) {
        //到数据库查询有没有该Cookie
        User user = userService.findUserByLoginToken(loginToken);
        if (user != null) {
            request.getSession().setAttribute("user", user);
            return true;
        } else {
            //没有该Cookie与之对应的用户(Cookie不匹配)
            Cookieutil.clearCookie(request, response, "loginToken");
            return false;
        }
    } else {
        //没有cookie、也没有登陆。是index请求获取用户信息，可以放行
        if (request.getRequestURI().contains("session")) {
            return true;
        }

        //没有cookie凭证
        response.sendRedirect("/login.html");
        return false;
    }
}
}
}

```

总结一下上面代码的思路：

- 用户登录时，验证用户的账户和密码
- 生成一个Token保存在数据库中，将Token写到Cookie中
- 将用户数据保存在Session中
- 请求时都会带上Cookie，检查有没有登录，如果已经登录则放行

## Cookie的作用是什么？和Session有什么区别？

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

**Cookie 一般用来保存用户信息** 比如①我们在 Cookie 中保存已经登录过得用户信息，下次访问网站的时候页面可以自动帮你登录的一些基本信息给填了；②一般的网站都会有保持登录也就是说下次你再访问网站的时候就不需要重新登录了，这是因为用户登录的时候我们可以存放了一个 Token 在 Cookie 中，下次登录的时候只需要根据 Token 值来查找用户即可(为了安全考虑，重新登录一般要将 Token 重写)；③登录一次网站后访问网站其他页面不需要重新登录。**Session 的主要作用就是通过服务端记录用户的状态**。典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了。

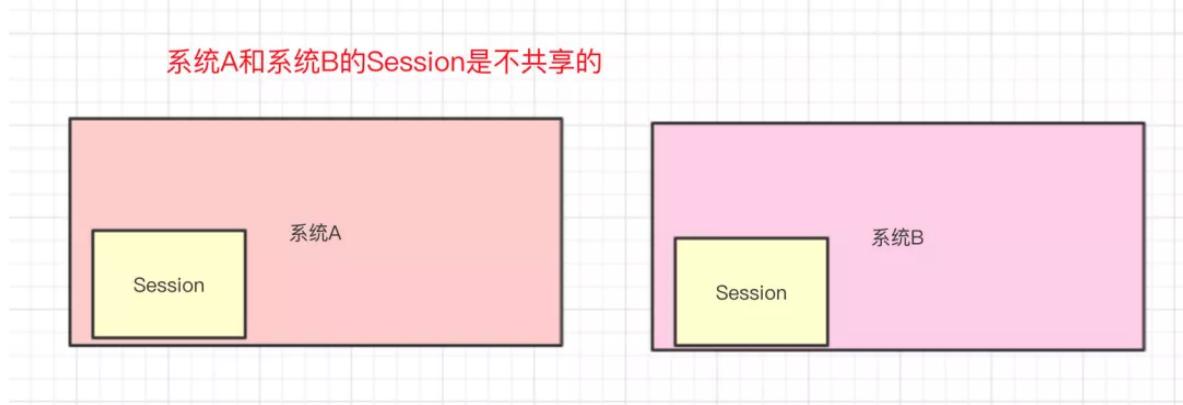
Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

Cookie 存储在客户端中，而Session存储在服务器上，相对来说 Session 安全性更高。如果使用 Cookie 的一些敏感信息不要写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

## 三、多系统登录的问题与解决

### 3.1 Session不共享问题

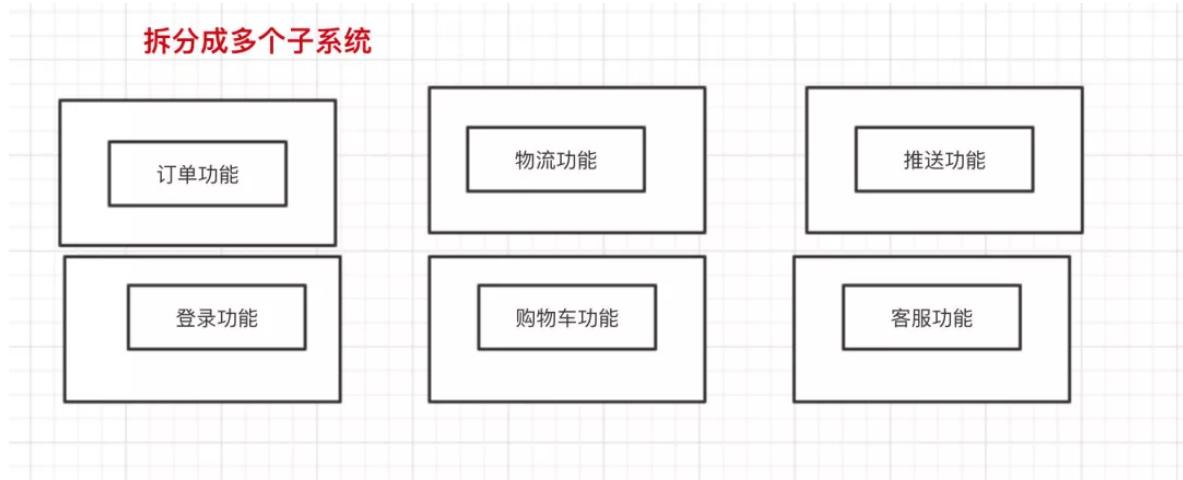
单系统登录功能主要是用Session保存用户信息来实现的，但我们清楚的是：多系统即可能有多个Tomcat，而Session是依赖当前系统的Tomcat，所以系统A的Session和系统B的Session是**不共享的**。



解决系统之间Session不共享问题有一下几种方案：

- Tomcat集群Session全局复制（集群内每个tomcat的session完全同步）【会影响集群的性能呢，不建议】
- 根据请求的IP进行**Hash映射**到对应的机器上（这就相当于请求的IP一直会访问同一个服务器）【如果服务器宕机了，会丢失了一大部分Session的数据，不建议】
- 把Session数据放在Redis中（使用Redis模拟Session）【建议】

我们可以将登录功能**单独抽取**出来，做一个子系统。



SSO (登录系统) 的逻辑如下：

```
// 登录功能(sso单独的服务)
@Override
public TaotaoResult login(String username, String password) throws Exception {

    //根据用户名查询用户信息
    TbUserExample example = new TbUserExample();
    Criteria criteria = example.createCriteria();
    criteria.andUsernameEqualTo(username);
```

```

List<TbUser> list = userMapper.selectByExample(example);
if (null == list || list.isEmpty()) {
    return TaotaoResult.build(400, "用户不存在");
}
//核对密码
TbUser user = list.get(0);
if
(!DigestUtils.md5DigestAsHex(password.getBytes()).equals(user.getPassword())) {
    return TaotaoResult.build(400, "密码错误");
}
//登录成功，把用户信息写入redis
//生成一个用户token
String token = UUID.randomUUID().toString();
jedisCluster.set(USER_TOKEN_KEY + ":" + token,
JsonUtils.objectToJson(user));
//设置session过期时间
jedisCluster.expire(USER_TOKEN_KEY + ":" + token, SESSION_EXPIRE_TIME);
return TaotaoResult.ok(token);
}

```

其他子系统登录时，请求SSO（登录系统）进行登录，将返回的token写到Cookie中，下次访问时则把Cookie带上：

```

public TaotaoResult login(String username, String password,
    HttpServletRequest request, HttpServletResponse response) {
    //请求参数
    Map<String, String> param = new HashMap<>();
    param.put("username", username);
    param.put("password", password);
    //登录处理
    String stringResult = HttpClientUtil.doPost(register_user_url +
USER_LOGIN_URL, param);
    TaotaoResult result = TaotaoResult.format(stringResult);
    //登录出错
    if (result.getStatus() != 200) {
        return result;
    }
    //登录成功后把取token信息，并写入cookie
    String token = (String) result.getData();
    //写入cookie
    CookieUtils.setCookie(request, response, "TT_TOKEN", token);
    //返回成功
    return result;
}

```

总结：

- SSO系统生成一个token，并将用户信息存到Redis中，并设置过期时间
- 其他系统请求SSO系统进行登录，得到SSO返回的token，写到Cookie中
- 每次请求时，Cookie都会带上，拦截器得到token，判断是否已经登录

到这里，其实我们会发现其实就两个变化：

- 将登陆功能抽取为一个系统（SSO），其他系统请求SSO进行登录
- 本来将用户信息存到Session，现在将用户信息存到Redis

## 3.2 Cookie跨域的问题

上面我们解决了Session不能共享的问题，但其实还有另一个问题。**Cookie是不能跨域的**

比如说，我们请求 `<https://www.google.com/>` 时，浏览器会自动把 `google.com` 的Cookie带过去给 `google` 的服务器，而不会把 `<https://www.baidu.com/>` 的Cookie带过去给 `google` 的服务器。

这就意味着，**由于域名不同**，用户向系统A登录后，系统A返回给浏览器的Cookie，用户再请求系统B的时候不会将系统A的Cookie带过去。

针对Cookie存在跨域问题，有几种解决方案：

1. 服务端将Cookie写到客户端后，客户端对Cookie进行解析，将Token解析出来，此后请求都把这个Token带上就行了
2. 多个域名共享Cookie，在写到客户端的时候设置Cookie的domain。
3. 将Token保存在SessionStorage中（不依赖Cookie就没有跨域的问题了）

到这里，我们已经可以实现单点登录了。

## 3.3 CAS原理

说到单点登录，就肯定见到这个名词：CAS (Central Authentication Service)，下面说说CAS是怎么搞的。

如果已经将登录单独抽取成系统出来，我们还能这样玩。现在我们有两个系统，分别是

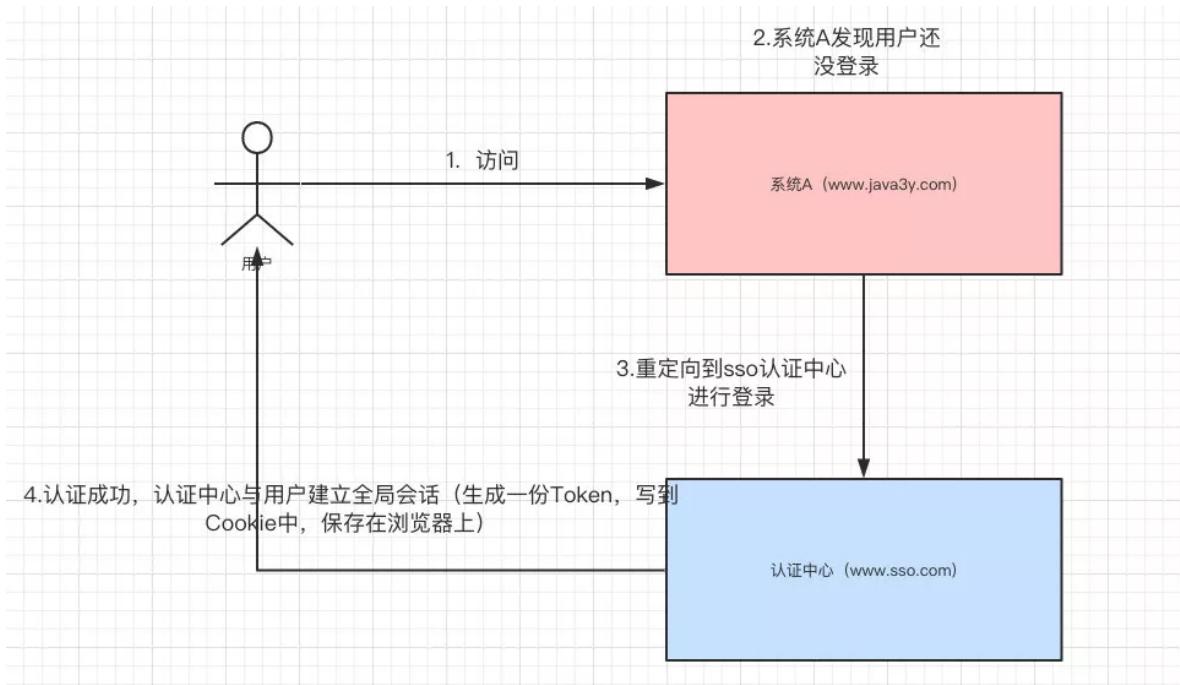
`www.java3y.com` 和 `www.java4y.com`，一个SSO `www.sso.com`



首先，用户想要访问系统A `www.java3y.com` 受限的资源(比如说购物车功能，购物车功能需要登录后才能访问)，系统A `www.java3y.com` 发现用户并没有登录，于是**重定向到sso认证中心，并将自己的地址作为参数**。请求的地址如下：

- `www.sso.com?service=www.java3y.com`

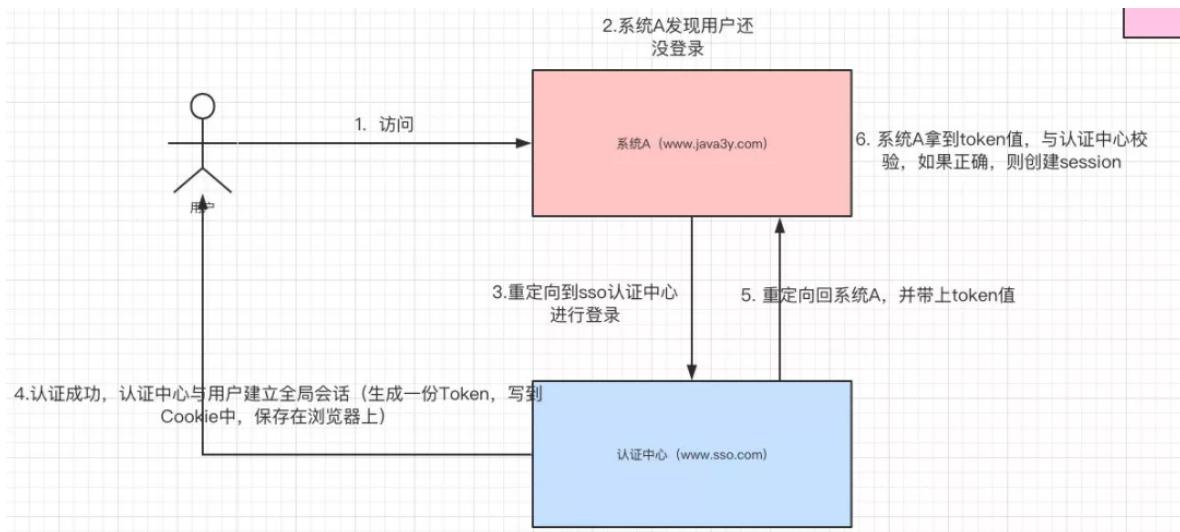
sso认证中心发现用户未登录，将用户引导至登录页面，用户进行输入用户名和密码进行登录，用户与认证中心建立**全局会话（生成一份Token，写到Cookie中，保存在浏览器上）**



随后，认证中心**重定向回系统A**，并把Token携带过去给系统A，重定向的地址如下：

- [www.java3y.com?token=xxxxxx](http://www.java3y.com?token=xxxxxx)

接着，系统A去sso认证中心验证这个Token是否正确，如果正确，则系统A和用户建立局部会话（**创建Session**）。至此，系统A和用户已经是登录状态了。



此时，用户想要访问系统B [www.java4y.com](http://www.java4y.com) 受限的资源(比如说订单功能，订单功能需要登录后才能访问)，系统B [www.java4y.com](http://www.java4y.com) 发现用户并没有登录，于是**重定向到sso认证中心，并将自己的地址作为参数**。请求的地址如下：

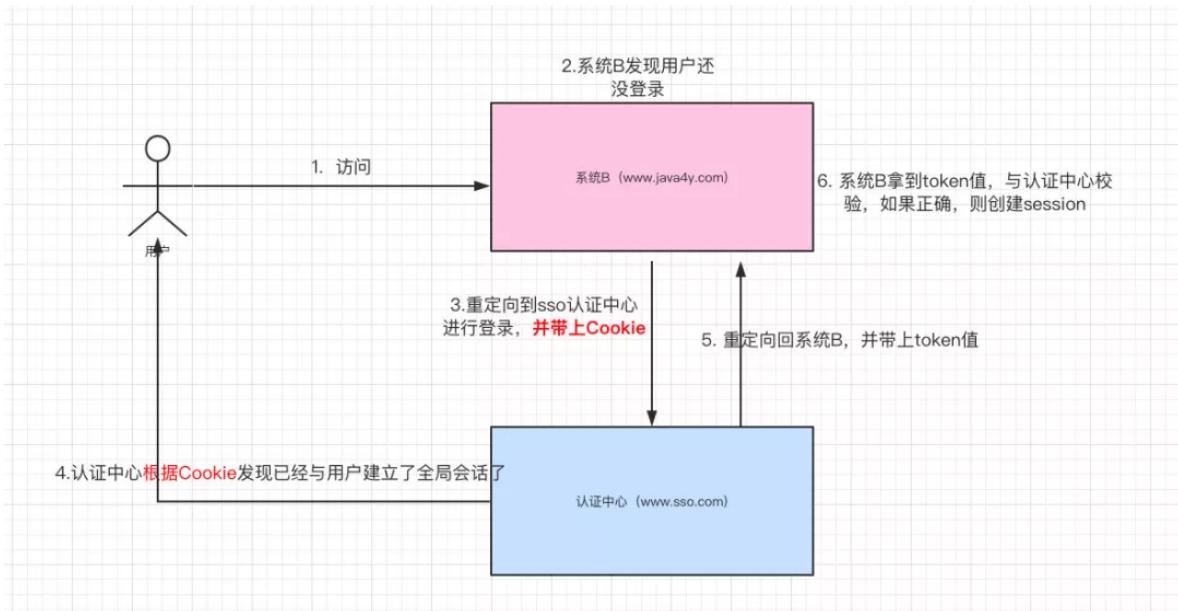
- [www.sso.com?service=www.java4y.com](http://www.sso.com?service=www.java4y.com)

注意，因为之前用户与认证中心 [www.sso.com](http://www.sso.com) 已经建立了全局会话（当时已经把Cookie保存到浏览器上了），所以这次系统B**重定向到认证中心 [www.sso.com](http://www.sso.com)**是可以带上Cookie的。

认证中心**根据带过来的Cookie**发现已经与用户建立了全局会话了，认证中心**重定向回系统B**，并把Token携带过去给系统B，重定向的地址如下：

- [www.java4y.com?token=xxxxxx](http://www.java4y.com?token=xxxxxx)

接着，系统B去sso认证中心验证这个Token是否正确，如果正确，则系统B和用户建立局部会话（**创建Session**）。到此，系统B和用户已经是登录状态了。



看到这里，其实SSO认证中心就类似一个**中转站**。

# 短网址

---

短网址（Short URL），是在形式上比较短的网址，通过映射关系跳转到原有的长网址。

目前已经有许多类似服务，借助短网址您可以用简短的网址替代原来冗长的网址，让使用者可以更容易的分享链接。

例如：

短网址：<http://t.cn/R3Krh36>

长网址：<https://blog.mimvp.com/article/25420.html>

生成短网址：<http://dwz.wailian.work/index.php>

看过新浪的短连接服务，发现后面主要有6个字符串组成，于是第一个想到的就是原来公司写的一个游戏激活码规则，也就是下面的算法2，

1) 26个大写字母 26小写字母，10个数字，随机生成6个然后插入数据库对应一个id，

2) 短连接跳转的时候，根据字符串查询到对应id，即可实现相应的跳转

62种字符组合成6位字符， $62^6=568$ 亿个组合数量，重复的概率是很小的

短链接的好处

1、内容需要；

2、用户友好；

3、便于管理。

为什么要这样做的，原因我想有这样几点：

1) 微博限制一条字数为140字，那么如果我们需要发一些连接上去，但是这个连接非常的长，以至于将近要占用我们内容的一半篇幅，这肯定是不能被允许的，所以短网址应运而生了。

2) 短网址在项目里可以很好的对开放级URL进行管理。有一部分网址可以会涵盖暴力、广告等信息，这样我们可以通过用户的举报，完全管理这些链接不出现在我们的应用中。因为同样的URL通过加密算法之后，得到的地址是一样的。

3) 我们可以对一系列的网址进行流量，点击等统计，挖掘出大多数用户的关注点，这样有利于我们对项目的后续工作更好的作出决策。

算法原理

- 算法一

1) 将长网址md5生成32位签名串，分为4段，每段8个字节；52c06085 c4529732 5433e0c7  
5b140565

2) 对这四段循环处理，取8个字节，将他看成16进制串与0x3fffffff(30位1)与操作，即前缀超过30位的字符串做忽略处理，直接舍弃掉了；

3) 这30位分成6段，每5位的数字作为字母表的索引取得特定字符，依次进行获得6位字符串；

4) 总的md5串可以获得4个6位串，取里面的任意一个就可作为这个长url的短url地址；

这种算法，虽然会生成4个，但是仍然存在重复几率

PHP 改进型

把固定长度6位，改进成动态调整的，如5、6、10、15位等，使其是30个质数之一

- 算法二

a-zA-Z0-9 这62位取6位组合，可产生 $62^6=568$ 亿个组合数量，把数字和字符组合做一定的映射，就可以产生唯一的字符串，如第62个组合就是aaaaaa9，第63个组合就是aaaaba，再利用洗牌算法，把原字符串打乱后保存，那么对应位置的组合字符串就会是无序的组合。

把长网址存入数据库，取返回的id，找出对应的字符串，例如返回ID为1，那么对应上面的字符串组合就是bbb，同理 ID为2时，字符串组合为bba，依次类推，直至到达62种组合后才会出现重复的可能，所以如果用上面的62个字符，任意取6个字符组合成字符串的话，你的数据存量达到500多亿后才会出现重复的可能。具体参看这里彻底完善新浪微博接口和超短URL算法，算法四可以算是此算法的一种实现，此算法一般不会重复，但是如果统计的话，就有很大问题，特别是对域名相关的统计，就抓虾了。

原理：指定长度，做多次循环，每次从长字符串里随机取出一位字符，组合成指定长度字符串即可

## 跳转原理

---

当我们生成短链接之后，只需要在数据库MySQL 或 NoSQL表中，存储原始链接与短链接的映射关系即可。

当我们访问短链接时，只需要从映射关系中找到原始链接，即可跳转到原始链接。

例如：

短网址：<http://t.cn/R3Krh36> (存储长链接的映射)

长网址：<https://blog.mimvp.com/article/25420.html>

# 分布式

---

## 分布式锁

---

在单机场景下，可以使用语言的内置锁来实现进程同步。但是在分布式场景下，需要同步的进程可能位于不同的节点上，那么就需要使用分布式锁。

阻塞锁通常使用互斥量来实现：

- 互斥量为 0 表示有其它进程在使用锁，此时处于锁定状态；
- 互斥量为 1 表示未锁定状态。

1 和 0 可以用一个整型值表示，也可以用某个数据是否存在表示。

## 数据库的唯一索引

获得锁时向表中插入一条记录，释放锁时删除这条记录。唯一索引可以保证该记录只被插入一次，那么就可以用这个记录是否存在来判断是否存于锁定状态。

优点：

- 易于理解实现

缺点：

- 锁没有失效时间，解锁失败的话其它进程无法再获得该锁。
- 只能是非阻塞锁，插入失败直接就报错了，无法重试。
- 不可重入，已经获得锁的进程也必须重新获取锁。

## Redis 的 SETNX 指令

使用 SETNX (set if not exist) 指令插入一个键值对，如果 Key 已经存在，那么会返回 False，否则插入成功并返回 True。

SETNX 指令和数据库的唯一索引类似，保证了只存在一个 Key 的键值对，那么可以用一个 Key 的键值对是否存在来判断是否存于锁定状态。

EXPIRE 指令可以为一个键值对设置一个过期时间，从而避免了数据库唯一索引实现方式中释放锁失败的问题。

优点：

- 吞吐量高
- 有锁失效自动删除机制，保证不会阻塞所有流程

缺点：

- 单点故障问题
- 锁超时问题：如果A拿到锁之后设置了超时时长，但是业务还未执行完成且锁已经被释放，此时其他进程就会拿到锁从而执行相同的业务
- 轮询获取锁状态方式太过低效

## Redis 的 RedLock 算法

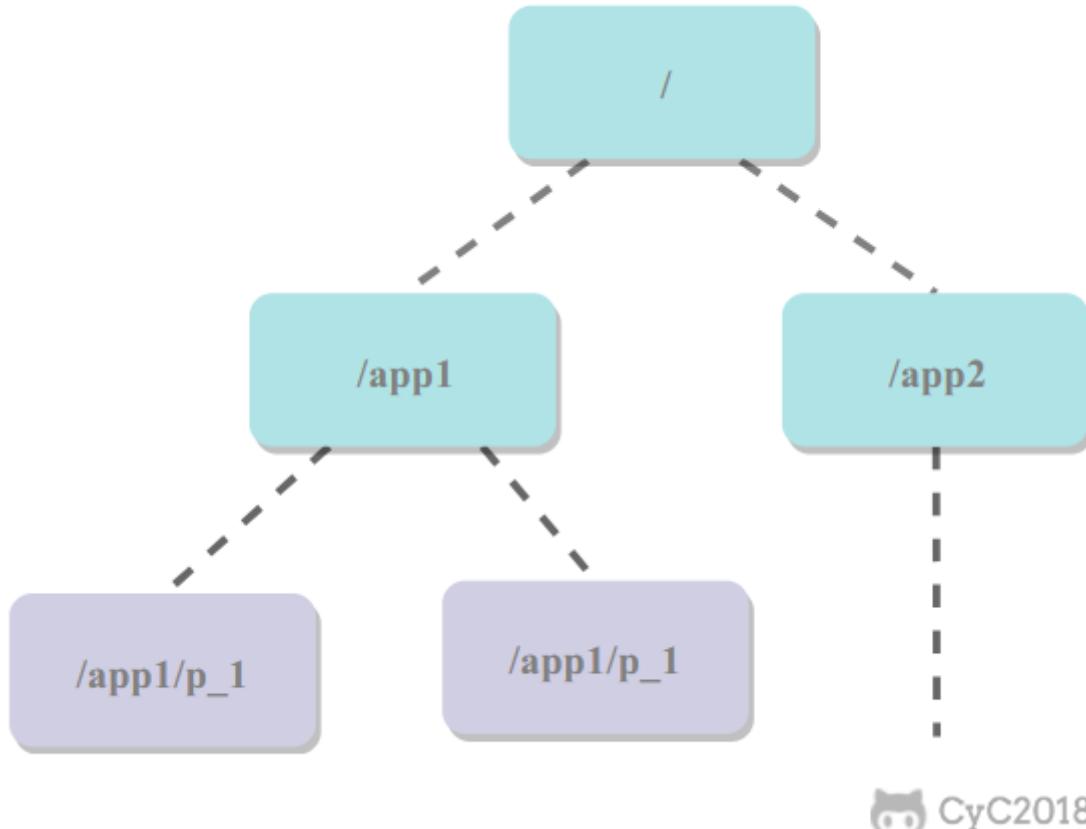
使用了多个 Redis 实例来实现分布式锁，这是为了保证在发生单点故障时仍然可用。

- 尝试从 N 个互相独立 Redis 实例获取锁；
- 计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且从大多数 ( $N / 2 + 1$ ) 实例上获取了锁，那么就认为锁获取成功了；
- 如果锁获取失败，就到每个实例上释放锁。

## Zookeeper 的有序节点

### 1. Zookeeper 抽象模型

Zookeeper 提供了一种树形结构的命名空间，/app1/p\_1 节点的父节点为 /app1。



### 2. 节点类型

- 永久节点：不会因为会话结束或者超时而消失；
- 临时节点：如果会话结束或者超时就会消失；
- 有序节点：会在节点名的后面加一个数字后缀，并且是有序的，例如生成的有序节点为 /lock/node-0000000000，它的下一个有序节点则为 /lock/node-0000000001，以此类推。

### 3. 监听器

为一个节点注册监听器，在节点状态发生改变时，会给客户端发送消息。

### 4. 分布式锁实现

- 创建一个锁目录 /lock；
- 当一个客户端需要获取锁时，在 /lock 下创建临时的且有序的子节点；
- 客户端获取 /lock 下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁；否则监听自己的前一个子节点，获得子节点的变更通知后重复此步骤直至获得锁；
- 执行业务代码，完成后，删除对应的子节点。

## 5. 会话超时

如果一个已经获得锁的会话超时了，因为创建的是临时节点，所以该会话对应的临时节点会被删除，其它会话就可以获得锁了。可以看到，Zookeeper 分布式锁不会出现数据库的唯一索引实现的分布式锁释放锁失败问题。

## 6. 羊群效应

一个节点未获得锁，只需要监听自己的前一个子节点，这是因为如果监听所有的子节点，那么任意一个子节点状态改变，其它所有子节点都会收到通知（羊群效应），而我们只希望它的后一个子节点收到通知。

优点：

- 解决锁超时问题。因为Zookeeper的写入都是顺序的，在一个节点创建之后，其他请求再次创建便会失败，同时可以对这个节点进行Watch，如果节点删除会通知其他节点抢占锁
- 能通过watch机制高效通知其他节点获取锁，避免惊群效应
- 有锁失效自动删除机制，保证不会阻塞所有流程

缺点：

- 性能不如Redis
- 强依赖zk，如果原来系统不用zk那就需要维护一套zk

# 分布式事务

指事务的操作位于不同的节点上，需要保证事务的 ACID 特性。

例如在下单场景下，库存和订单如果不在同一个节点上，就涉及分布式事务。

## 2PC

两阶段提交 (Two-phase Commit, 2PC)，通过引入协调者 (Coordinator) 来协调参与者的行，并最终决定这些参与者是否要真正执行事务。

### 1. 运行过程

#### 1.1 准备阶段

协调者询问参与者事务是否执行成功，参与者发回事务执行结果。



#### 1.2 提交阶段

如果事务在每个参与者上都执行成功，事务协调者发送通知让参与者提交事务；否则，协调者发送通知让参与者回滚事务。

需要注意的是，在准备阶段，参与者执行了事务，但是还未提交。只有在提交阶段接收到协调者发来的通知后，才进行提交或者回滚。



CyC2018

## 2. 存在的问题

### 2.1 同步阻塞

所有事务参与者在等待其它参与者响应的时候都处于同步阻塞状态，无法进行其它操作。

### 2.2 单点问题

协调者在 2PC 中起到非常大的作用，发生故障将会造成很大影响。特别是在阶段二发生故障，所有参与者会一直等待，无法完成其它操作。

### 2.3 数据不一致

在阶段二，如果协调者只发送了部分 Commit 消息，此时网络发生异常，那么只有部分参与者接收到 Commit 消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。

### 2.4 太过保守

任意一个节点失败就会导致整个事务失败，没有完善的容错机制。

## 3PC

3PC，三阶段提交协议，是2PC的改进版本，即将事务的提交过程分为CanCommit、PreCommit、do Commit三个阶段来进行处理。

### 阶段1：CanCommit

- 1、协调者向所有参与者发出包含事务内容的CanCommit请求，询问是否可以提交事务，并等待所有参与者答复。
- 2、参与者收到CanCommit请求后，如果认为可以执行事务操作，则反馈YES并进入预备状态，否则反馈NO。

### 阶段2：PreCommit

此阶段分两种情况：

- 1、所有参与者均反馈YES，即执行事务预提交。
- 2、任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务。

事务预提交：（所有参与者均反馈YES时）

- 1、协调者向所有参与者发出PreCommit请求，进入准备阶段。
- 2、参与者收到PreCommit请求后，执行事务操作，将Undo和Redo信息记入事务日志中（但不提交事务）。
- 3、各参与者向协调者反馈Ack响应或No响应，并等待最终指令。

中断事务：（任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈时）

- 1、协调者向所有参与者发出abort请求。
- 2、无论收到协调者发出的abort请求，或者在等待协调者请求过程中出现超时，参与者均会中断事务。

### 阶段3：do Commit

此阶段也存在两种情况：

- 1、所有参与者均反馈Ack响应，即执行真正的事务提交。
- 2、任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务。

提交事务：（所有参与者均反馈Ack响应时）

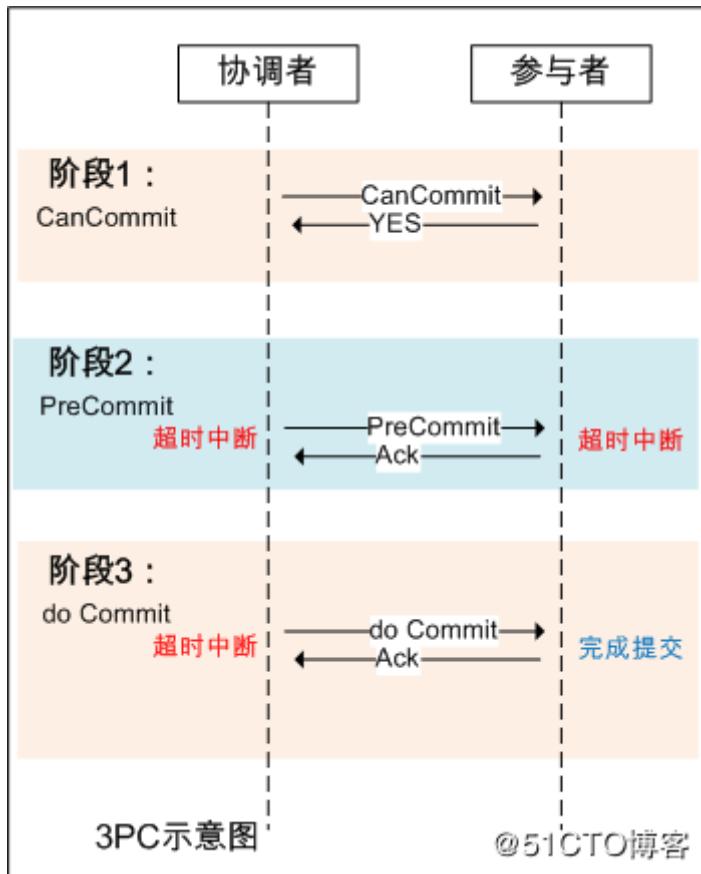
- 1、如果协调者处于工作状态，则向所有参与者发出do Commit请求。
- 2、参与者收到do Commit请求后，会正式执行事务提交，并释放整个事务期间占用的资源。
- 3、各参与者向协调者反馈Ack完成的消息。
- 4、协调者收到所有参与者反馈的Ack消息后，即完成事务提交。

中断事务：（任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈时）

- 1、如果协调者处于工作状态，向所有参与者发出abort请求。
- 2、参与者使用阶段1中的Undo信息执行回滚操作，并释放整个事务期间占用的资源。
- 3、各参与者向协调者反馈Ack完成的消息。
- 4、协调者收到所有参与者反馈的Ack消息后，即完成事务中断。

注意：进入阶段三后，无论协调者出现问题，或者协调者与参与者网络出现问题，都会导致参与者无法接收到协调者发出的do Commit请求或abort请求。此时，参与者都会在等待超时之后，继续执行事务提交。

附示意图如下：



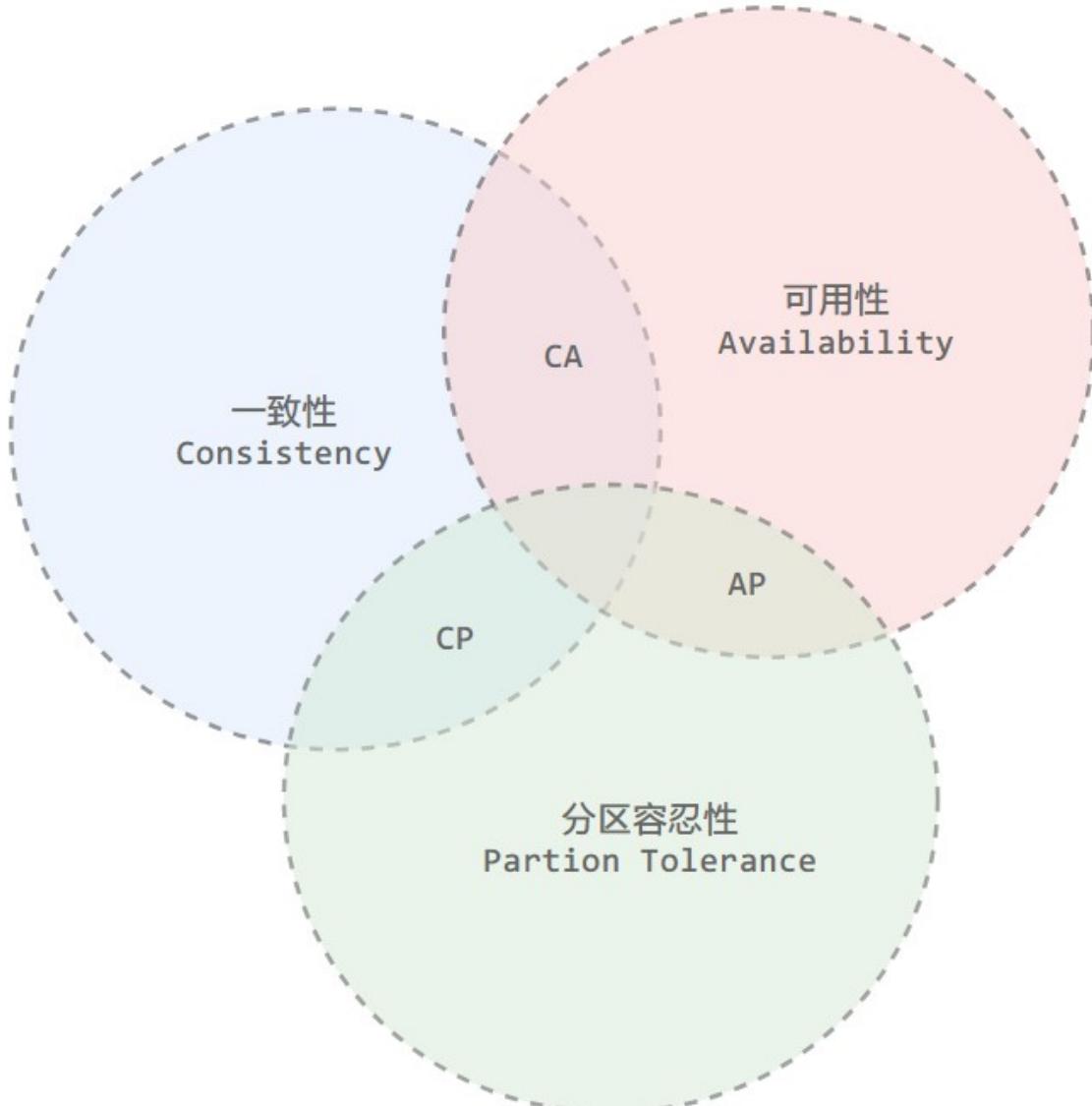
## 3PC的优点和缺陷

优点：降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题，阶段3中协调者出现问题时，参与者会继续提交事务。

缺陷：脑裂问题依然存在，即在参与者收到PreCommit请求后等待最终指令，如果此时协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

## CAP

分布式系统不可能同时满足一致性（C: Consistency）、可用性（A: Availability）和分区容忍性（P: Partition Tolerance），最多只能同时满足其中两项。



 CyC2018

## 一致性

一致性指的是多个数据副本是否能保持一致的特性，在一致性的条件下，系统在执行数据更新操作之后能够从一致性状态转移到另一个一致性状态。

对系统的一个数据更新成功之后，如果所有用户都能够读取到最新的值，该系统就被认为具有强一致性。

## 可用性

可用性指分布式系统在面对各种异常时可以提供正常服务的能力，可以用系统可用时间占总时间的比值来衡量，4个9的可用性表示系统99.99%的时间是可用的。

在可用性条件下，要求系统提供的服务一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

## 分区容忍性

网络分区指分布式系统中的节点被划分为多个区域，每个区域内部可以通信，但是区域之间无法通信。

在分区容忍性条件下，分布式系统在遇到任何网络分区故障的时候，仍然需要能对外提供一致性和可用性的服务，除非是整个网络环境都发生了故障。

## 权衡

在分布式系统中，分区容忍性必不可少，因为需要总是假设网络是不可靠的。因此，CAP 理论实际上是要在可用性和一致性之间做权衡。

可用性和一致性往往是冲突的，很难使它们同时满足。在多个节点之间进行数据同步时，

- 为了保证一致性 (CP)，不能访问未同步完成的节点，也就失去了部分可用性；
- 为了保证可用性 (AP)，允许读取所有节点的数据，但是数据可能不一致。

## BASE

---

BASE 是基本可用 (Basically Available)、软状态 (Soft State) 和最终一致性 (Eventually Consistent) 三个短语的缩写。

BASE 理论是对 CAP 中一致性和可用性权衡的结果，它的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

### 基本可用

指分布式系统在出现故障的时候，保证核心可用，允许损失部分可用性。

例如，电商在做促销时，为了保证购物系统的稳定性，部分消费者可能会被引导到一个降级的页面。

### 软状态

指允许系统中的数据存在中间状态，并认为该中间状态不会影响系统整体可用性，即允许系统不同节点的数据副本之间进行同步的过程存在时延。

### 最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能达到一致的状态。

ACID 要求强一致性，通常运用在传统的数据库系统上。而 BASE 要求最终一致性，通过牺牲强一致性来达到可用性，通常运用在大型分布式系统中。

在实际的分布式场景中，不同业务单元和组件对一致性的要求是不同的，因此 ACID 和 BASE 往往会结合在一起使用。

## Paxos

---

用于达成共识性问题，即对多个节点产生的值，该算法能保证只选出唯一一个值。

主要有三类节点：

- 提议者 (Proposer)：提议一个值；
- 接受者 (Acceptor)：对每个提议进行投票；
- 告知者 (Learner)：被告知投票的结果，不参与投票过程。

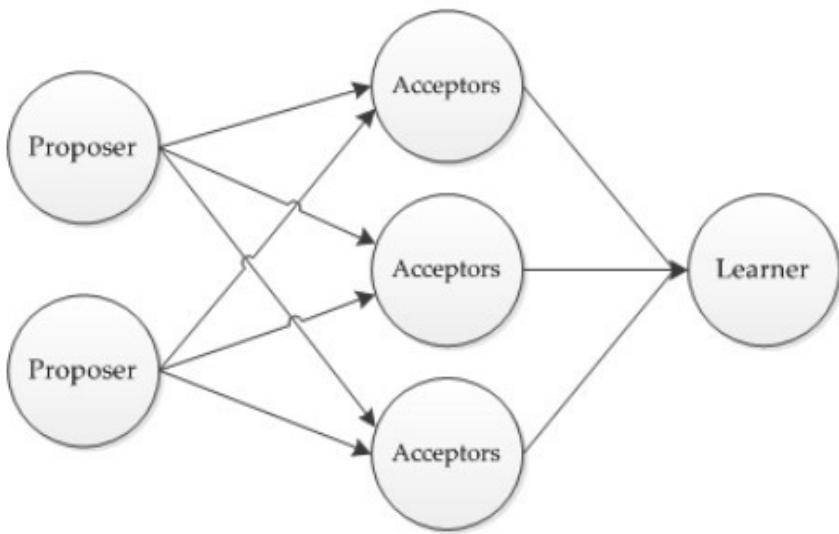


Figure 1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

## 执行过程

规定一个提议包含两个字段： $[n, v]$ ，其中  $n$  为序号（具有唯一性）， $v$  为提议值。

### 1. Prepare 阶段

下图演示了两个 Proposer 和三个 Acceptor 的系统中运行该算法的初始过程，每个 Proposer 都会向所有 Acceptor 发送 Prepare 请求。

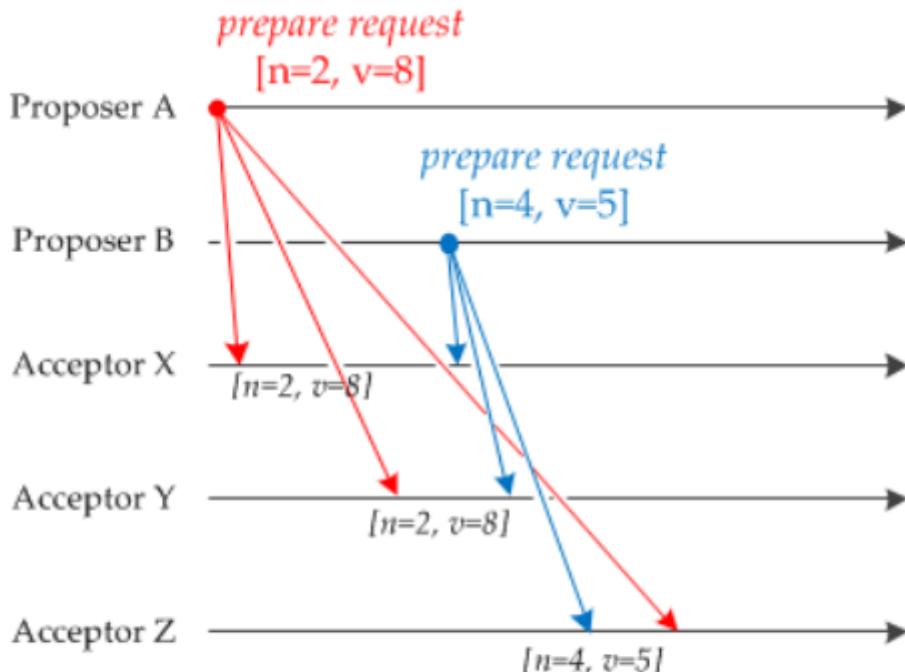


Figure 2: Paxos. Proposers A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.

当 Acceptor 接收到一个 Prepare 请求，包含的提议为  $[n_1, v_1]$ ，并且之前还未接收过 Prepare 请求，那么发送一个 Prepare 响应，设置当前接收到的提议为  $[n_1, v_1]$ ，并且保证以后不会再接受序号小于  $n_1$  的提议。

如下图，Acceptor X 在收到  $[n=2, v=8]$  的 Prepare 请求时，由于之前没有接收过提议，因此就发送一个  $[no previous]$  的 Prepare 响应，设置当前接收到的提议为  $[n=2, v=8]$ ，并且保证以后不会再接受序号小于 2 的提议。其它的 Acceptor 类似。

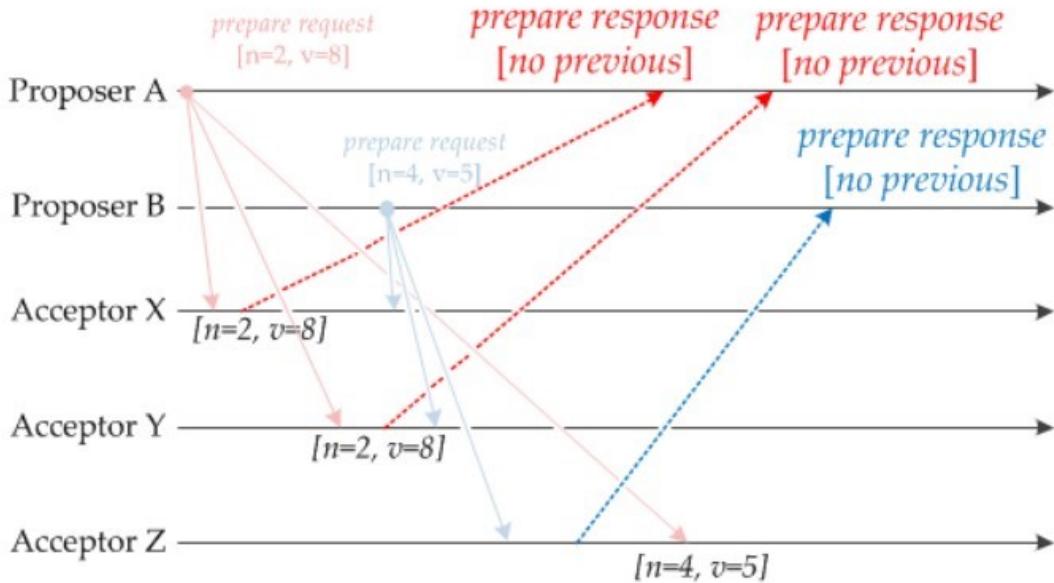


Figure 3: Paxos. Each acceptor responds to the first prepare request message that it receives.

如果 Acceptor 接收到一个 Prepare 请求，包含的提议为  $[n_2, v_2]$ ，并且之前已经接收过提议  $[n_1, v_1]$ 。如果  $n_1 > n_2$ ，那么就丢弃该提议请求；否则，发送 Prepare 响应，该 Prepare 响应包含之前已经接收过的提议  $[n_1, v_1]$ ，设置当前接收到的提议为  $[n_2, v_2]$ ，并且保证以后不会再接受序号小于  $n_2$  的提议。

如下图，Acceptor Z 收到 Proposer A 发来的  $[n=2, v=8]$  的 Prepare 请求，由于之前已经接收过  $[n=4, v=5]$  的提议，并且  $n > 2$ ，因此就抛弃该提议请求；Acceptor X 收到 Proposer B 发来的  $[n=4, v=5]$  的 Prepare 请求，因为之前接收到的提议为  $[n=2, v=8]$ ，并且  $2 \leq 4$ ，因此就发送  $[n=2, v=8]$  的 Prepare 响应，设置当前接收到的提议为  $[n=4, v=5]$ ，并且保证以后不会再接受序号小于 4 的提议。Acceptor Y 类似。

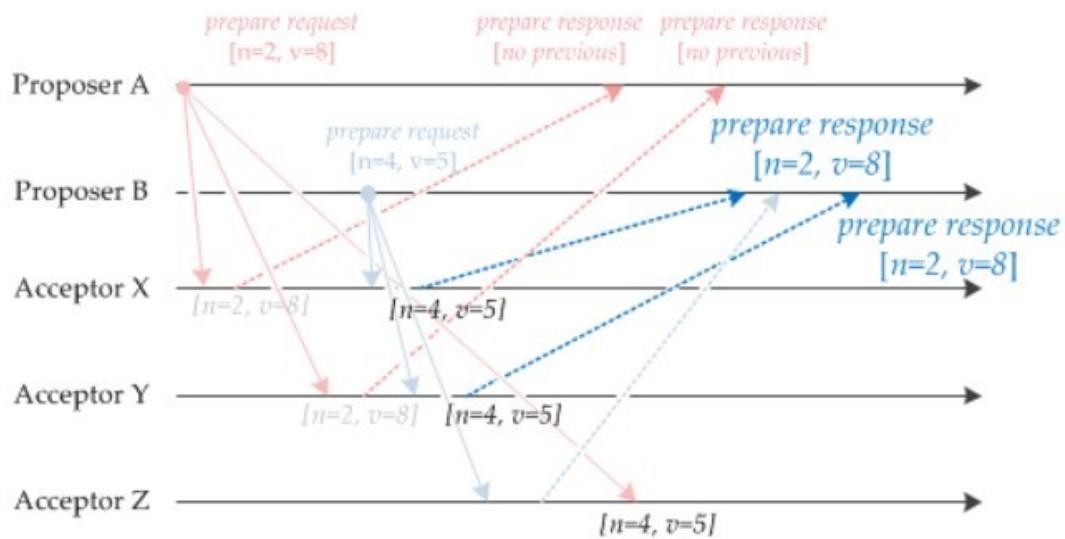


Figure 4: Paxos. Acceptor Z ignores proposer A's request because it has already seen a higher numbered proposal ( $4 > 2$ ). Acceptors X and Y respond to proposer B's request with the previous highest request that they acknowledged, and a promise to ignore any lower numbered proposals.

## 2. Accept 阶段

当一个 Proposer 接收到超过一半 Acceptor 的 Prepare 响应时，就可以发送 Accept 请求。

Proposer A 接收到两个 Prepare 响应之后，就发送  $[n=2, v=8]$  Accept 请求。该 Accept 请求会被所有 Acceptor 丢弃，因为此时所有 Acceptor 都保证不接受序号小于 4 的提议。

Proposer B 过后也收到了两个 Prepare 响应，因此也开始发送 Accept 请求。需要注意的是，Accept 请求的  $v$  需要取它收到的最大提议编号对应的  $v$  值，也就是 8。因此它发送  $[n=4, v=8]$  的 Accept 请求。

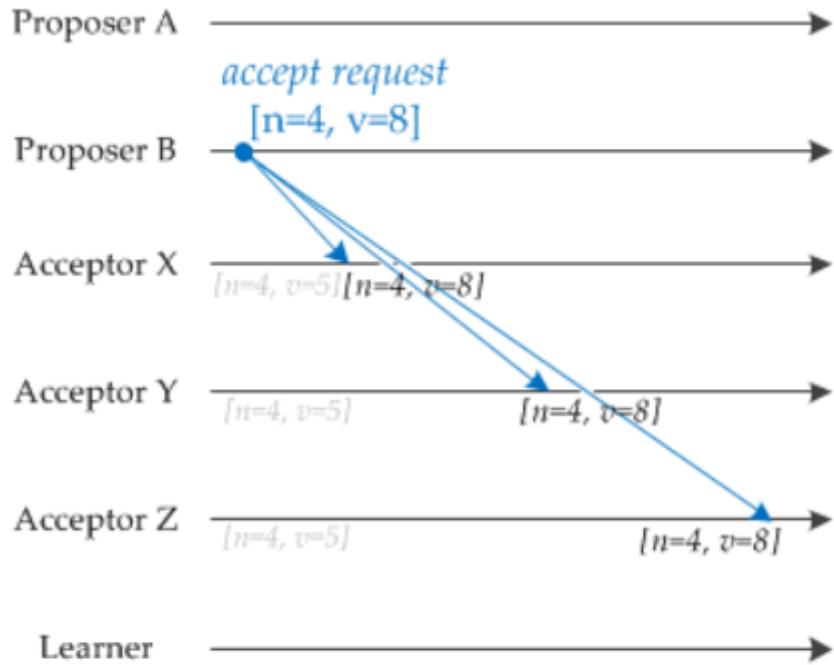
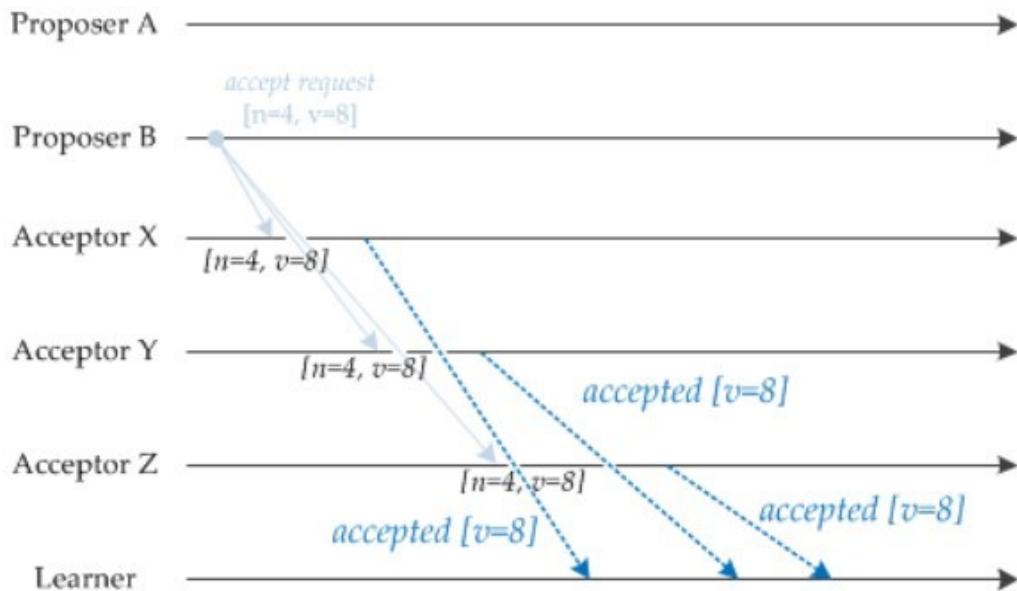


Figure 5: Paxos. Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen (8, from [n=2, v=8]

### 3. Learn 阶段

Acceptor 接收到 Accept 请求时，如果序号大于等于该 Acceptor 承诺的最小序号，那么就发送 Learn 提议给所有的 Learner。当 Learner 发现有大多数的 Acceptor 接收了某个提议，那么该提议的提议值就被 Paxos 选择出来。



## 约束条件

### 1. 正确性

指只有一个提议值会生效。

因为 Paxos 协议要求每个生效的提议被多数 Acceptor 接收，并且 Acceptor 不会接受两个不同的提议，因此可以保证正确性。

## 2. 可终止性

指最后总会有一个提议生效。

Paxos 协议能够让 Proposer 发送的提议朝着能被大多数 Acceptor 接受的那个提议靠拢，因此能够保证可终止性。

# Raft

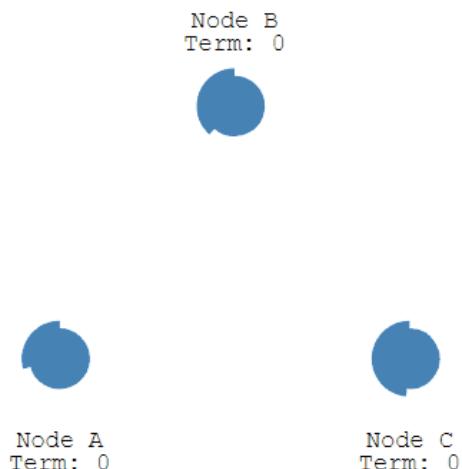
---

Raft 也是分布式一致性协议，主要是用来竞选主节点。

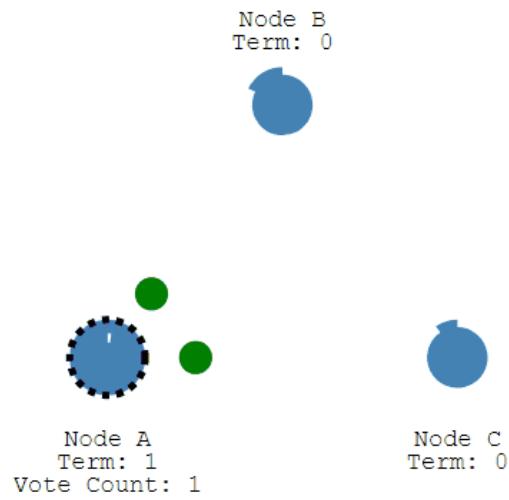
## 单个 Candidate 的竞选

有三种节点：Follower、Candidate 和 Leader。Leader 会周期性的发送心跳包给 Follower。每个 Follower 都设置了一个随机的竞选超时时间，一般为 150ms~300ms，如果在这个时间内没有收到 Leader 的心跳包，就会变成 Candidate，进入竞选阶段。

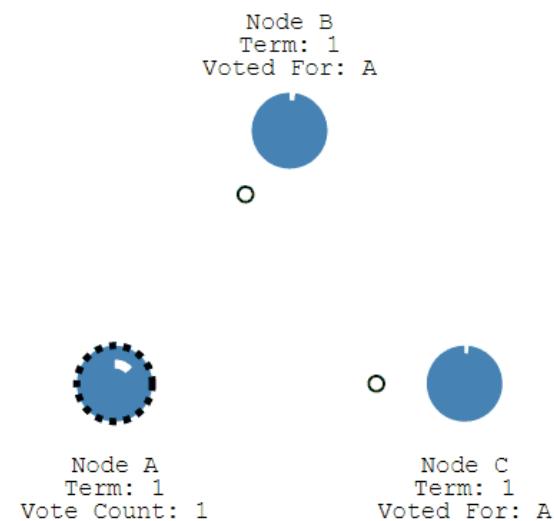
- 下图展示一个分布式系统的最初阶段，此时只有 Follower 没有 Leader。Node A 等待一个随机的竞选超时时间之后，没收到 Leader 发来的心跳包，因此进入竞选阶段。



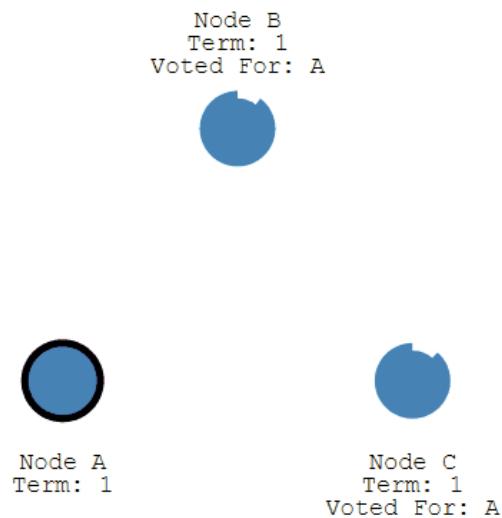
- 此时 Node A 发送投票请求给其它所有节点。



- 其它节点会对请求进行回复，如果超过一半的节点回复了，那么该 Candidate 就会变成 Leader。

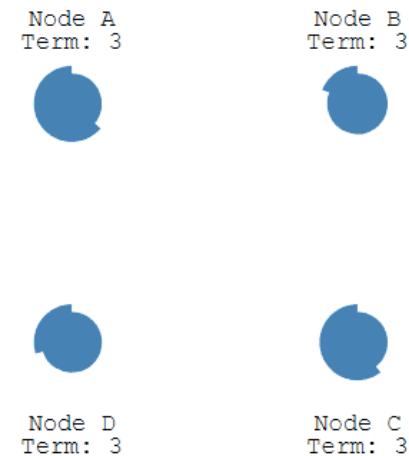


- 之后 Leader 会周期性地发送心跳包给 Follower，Follower 接收到心跳包，会重新开始计时。

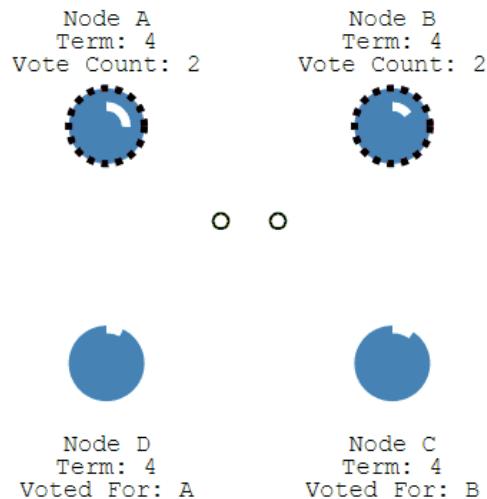


## 多个 Candidate 竞选

- 如果有多个 Follower 成为 Candidate，并且所获得票数相同，那么就需要重新开始投票。例如下图中 Node B 和 Node D 都获得两票，需要重新开始投票。



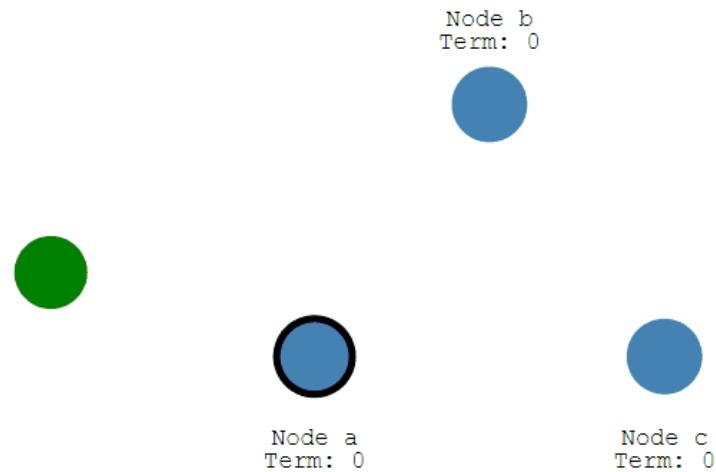
- 由于每个节点设置的随机竞选超时时间不同，因此下一次再次出现多个 Candidate 并获得同样票数的概率很低。



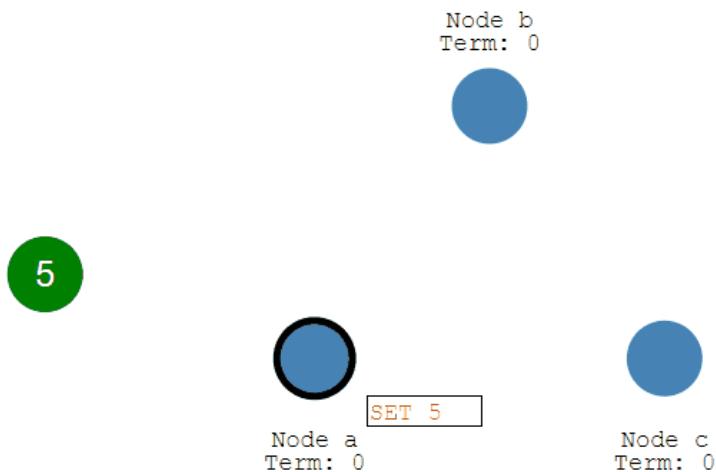
]

## 数据同步

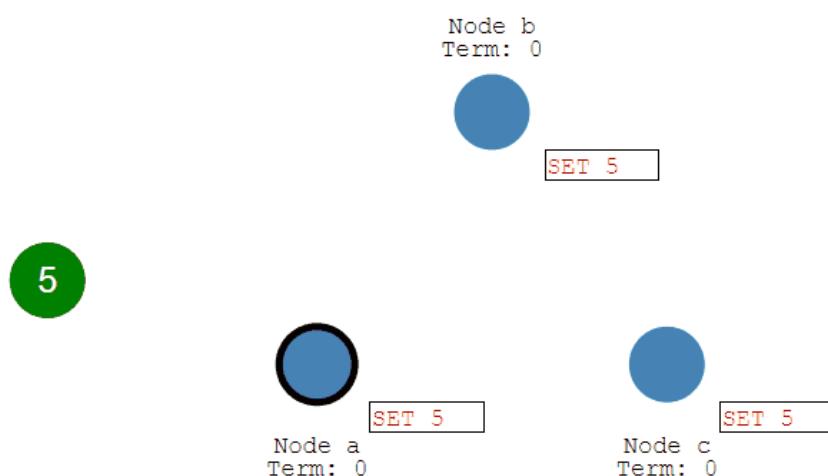
- 来自客户端的修改都会被传入 Leader。注意该修改还未被提交，只是写入日志中。



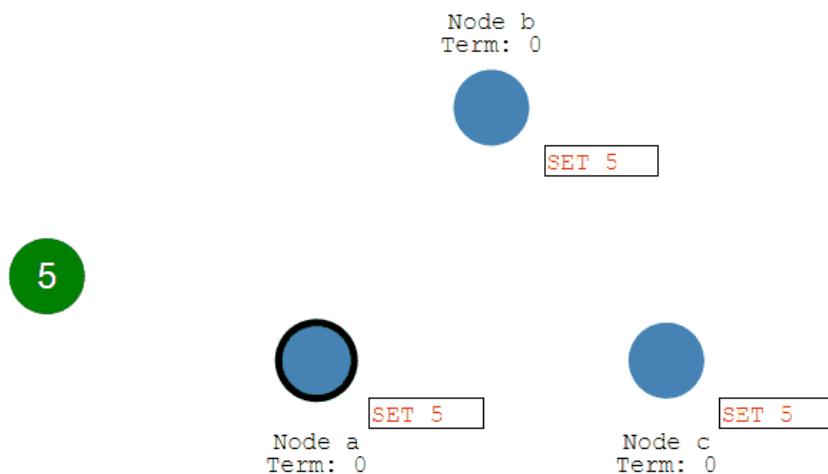
- Leader 会把修改复制到所有 Follower。



- Leader 会等待大多数的 Follower 也进行了修改，然后才将修改提交。



- 此时 Leader 会通知的所有 Follower 让它们也提交修改，此时所有节点的值达成一致。



# 高并发系统

---

其实所谓的高并发，如果你要理解这个问题呢，其实就得从高并发的根源出发，为啥会有高并发？为啥高并发就很牛逼？

我说的浅显一点，很简单，就是因为刚开始系统都是连接数据库的，但是要知道数据库支撑到每秒并发两三千的时候，基本就快完了。所以才有说，很多公司，刚开始干的时候，技术比较 low，结果业务发展太快，有的时候系统扛不住压力就挂了。

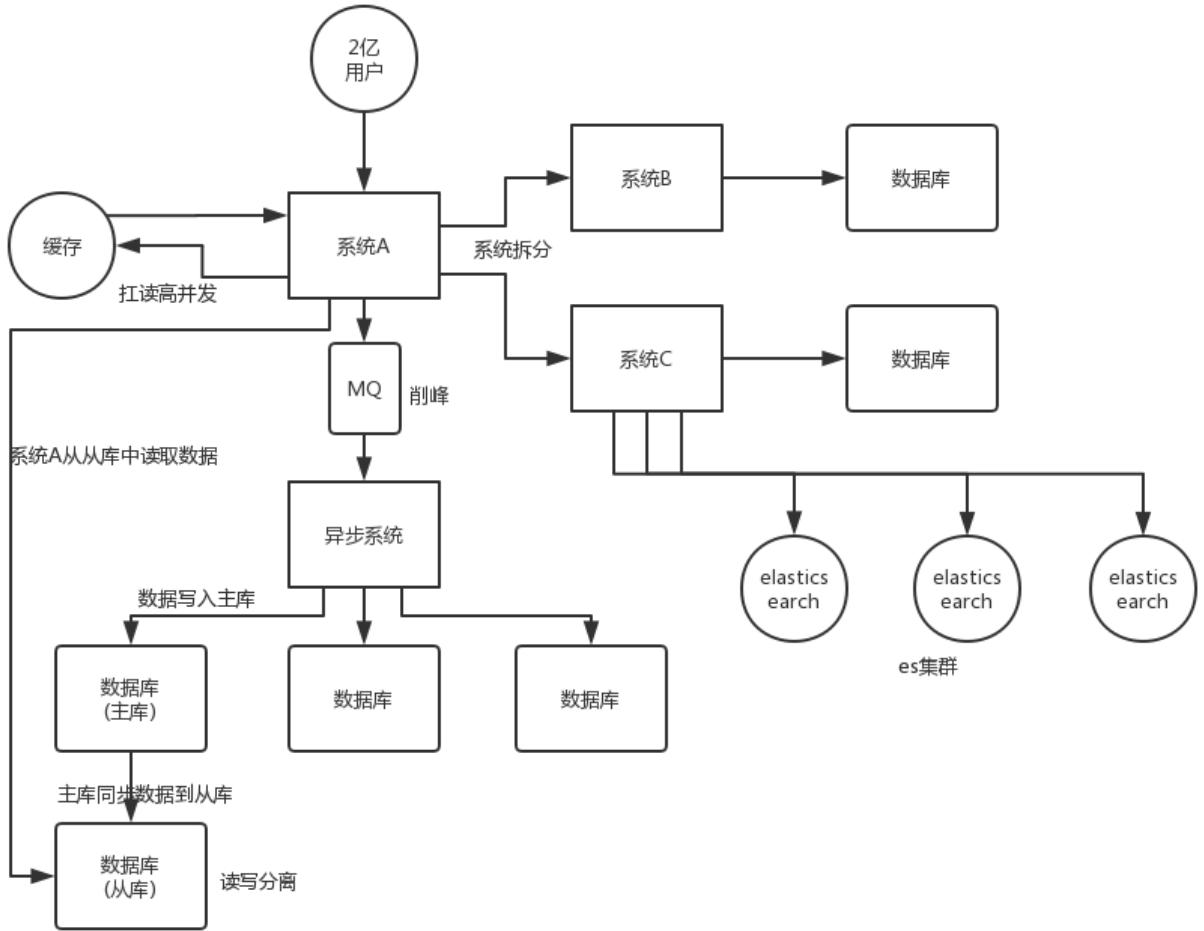
当然会挂了，凭什么不挂？你数据库如果瞬间承载每秒 5000/8000，甚至上万的并发，一定会宕机，因为比如 mysql 就压根儿扛不住这么高的并发量。

所以为啥高并发牛逼？就是因为现在用互联网的人越来越多，很多 app、网站、系统承载的都是高并发请求，可能高峰期每秒并发量几千，很正常的。如果是什么双十一之类的，每秒并发几万几十万都有可能。

那么如此之高的并发量，加上原本就如此之复杂的业务，咋玩儿？真正厉害的，一定是在复杂业务系统里玩儿过高并发架构的人，但是你没有，那么我给你说一下你该怎么回答这个问题：

可以分为以下 6 点：

- 系统拆分
- 缓存
- MQ
- 分库分表
- 读写分离
- ElasticSearch



## 系统拆分

将一个系统拆分为多个子系统，用 dubbo 来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以扛高并发么。

## 缓存

缓存，必须得用缓存。大部分的高并发场景，都是**读多写少**，那你完全可以在数据库和缓存里都写一份，然后读的时候走缓存不就得了吗。毕竟人家 redis 轻轻松松单机几万的并发。所以你可以考虑考虑你的项目里，那些承载主要请求的**读场景**，**怎么用缓存来抗高并发**。

## MQ

MQ，必须得用 MQ。可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改，疯了。那高并发绝对搞挂你的系统，你要用 redis 来承载写那肯定不行，人家是缓存，数据随时就被 LRU 了，数据格式还无比简单，没有事务支持。所以该用 mysql 还得用 mysql 啊。那你咋办？用 MQ 吧，大量的写请求灌入 MQ 里，排队慢慢玩儿，**后边系统消费后慢慢写**，控制在 mysql 承载范围之内。所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用 MQ 来异步写，提升并发性。MQ 单机抗几万并发也是 ok 的，这个之前还特意说过。

## 分库分表

分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表**拆分为多个表**，每个表的数据量保持少一点，提高sql跑的性能。

## 读写分离

读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，**主库写入，从库读取**，搞一个读写分离。读流量太多的时候，还可以加更多的从库。

## ElasticSearch

Elasticsearch，简称es。es是分布式的，可以随便扩容，分布式天然就可以支撑高并发，因为动不动就可以扩容加机器来扛更高的并发。那么一些比较简单的查询、统计类的操作，可以考虑用es来承载，还有一些全文搜索类的操作，也可以考虑用es来承载。

上面的6点，基本就是高并发系统肯定要干的一些事儿，大家可以仔细结合之前讲过的知识考虑一下，到时候你可以系统的把这块阐述一下，然后每个部分要注意哪些问题，之前都讲过了，你都可以阐述阐述，表明你对这块是有点积累的。

说句实话，毕竟你真正厉害的一点，不是在于弄明白一些技术，或者大概知道一个高并发系统应该长什么样？其实实际上在真正的复杂的业务系统里，做高并发要远远比上面提到的点要复杂几十倍到上百倍。你需要考虑：哪些需要分库分表，哪些不需要分库分表，单库单表跟分库分表如何join，哪些数据要放到缓存里去，放哪些数据才可以扛住高并发的请求，你需要完成对一个复杂业务系统的分析之后，然后逐步逐步的加入高并发的系统架构的改造，这个过程是无比复杂的，一旦做过一次，并且做好了，你在这个市场上就会非常的吃香。

其实大部分公司，真正看重的，不是说你掌握高并发相关的一些基本的架构知识，架构中的一些技术，RocketMQ、Kafka、Redis、Elasticsearch，高并发这一块，你了解了，也只能是次一等的人才。对一个有几十万行代码的复杂的分布式系统，一步一步架构、设计以及实践过高并发架构的人，这个经验是难能可贵的。

# 海量数据处理面试

## 海量日志数据，提取出某日访问百度次数最多的那个IP

- 分而治之：数据量太大，内存受限。把大文件划分为小文件（取模映射）
- 哈希统计：HashMap<IP, count>
- 取最大值

**搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门），请你统计最热门的10个查询串，要求使用的内存不能超过1G**

由上面第1题，我们知道，数据大则划为小的，但如果数据规模比较小，能一次性装入内存呢？比如这第2题，虽然有一千万个Query，但是由于重复度比较高，因此事实上只有300万的Query，每个Query 255Byte，因此我们可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，Hash Table 绝对是我们优先的选择。所以我们摒弃分而治之/hash映射的方法，直接上hash统计，然后排序。So，

- hash统计：先对这批海量数据预处理(维护一个Key为Query字串，Value为该Query出现次数的HashTable，即hash\_map(Query, Value)，每次读取一个Query，如果该字串不在Table中，那么加入该字串，并且将Value值设为1；如果该字串在Table中，那么将该字串的计数加一即可。最终我们在O(N)的时间复杂度内用Hash表完成了统计；
- 堆排序：第二步、借助堆这个数据结构，找出Top K，时间复杂度为N'logK。即借助堆结构，我们可以在log量级的时间内查找和调整/移动。因此，维护一个K(该题目中是10)大小的小根堆，然后遍历300万的Query，分别和根元素进行对比所以，我们最终的时间复杂度是：O (N) + N'\*O (logK) ， (N为1000万， N'为300万)。

**有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词**

- 分而治之/hash映射：顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。
- hash统计：对每个小文件，采用trie树/hash\_map等统计每个文件中出现的词以及相应的频率。
- 堆/归并排序：取出出现频率最大的100个词（可以用含100个结点的最小堆），并把100个词及相应的频率存入文件，这样又得到了5000个文件。最后就是把这5000个文件进行归并（类似于归并排序）的过程了。

**海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10**

- 堆排序：在每台电脑上求出TOP10，可以采用包含10个元素的堆完成（TOP10小，用最大堆，TOP10大，用最小堆）。比如求TOP10大，我们首先取前10个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是TOP10大。
- 求出每台电脑上的TOP10后，然后把这100台电脑上的TOP10组合起来，共1000个数据，再利用上面类似的方法求出TOP10就可以了。

## 有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序

直接上：

- hash映射：顺序读取10个文件，按照 $\text{hash(query)} \% 10$ 的结果将query写入到另外10个文件（记为）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。
- hash统计：找一台内存2G左右的机器，依次对用 $\text{hash\_map(query, query\_count)}$ 来统计每个query出现的次数。注： $\text{hash\_map(query, query\_count)}$ 是用来统计每个query的出现次数，不是存储他们的值，出现一次，则 $\text{count}+1$ 。
- 堆/快速/归并排序：利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的 $\text{query\_cout}$ 输出到文件中。这样得到了10个排好序的文件（记为）。对这10个文件进行归并排序（内排序与外排序相结合）。

除此之外，此题还有以下两个方法：

方案2：一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash\_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案3：与方案1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

## 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url

可以估计每个文件的大小为 $5G \times 64 = 320G$ ，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

- 分而治之/hash映射：遍历文件a，对每个url求取 $\text{hash(url)} \% 1000$ ，然后根据所取得的值将url分别存储到1000个小文件（记为 $a_0, a_1, \dots, a_{999}$ ）中。这样每个小文件的大约300M。遍历文件b，采取和a相同的方式将url分别存储到1000小文件中（记为 $b_0, b_1, \dots, b_{999}$ ）。这样处理后，所有可能相同的url都在对应的小文件（ $a_0 \text{ vs } b_0, a_1 \text{ vs } b_1, \dots, a_0 \text{ vs } b_{999}$ ）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。
- hash统计：求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

## 怎么在海量数据中找出重复次数最多的一个？

先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

## **上千万或上亿数据（有重复），统计其中出现次数最多的N个数据。**

---

上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash\_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用堆机制完成。

## **一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。**

---

这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是 $O(n \cdot l_e)$  ( $l_e$ 表示单词的平均长度)。然后是找出出现最频繁的前10个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n \cdot \lg 10)$ 。所以总的时间复杂度，是 $O(n \cdot l_e)$ 与 $O(n \cdot \lg 10)$ 中较大的哪一个。

## **2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数**

---

方案一：整数个数为 $2^{32}$ ，内存中每一位表示一个整数，需要约500M空间。

方案二（判断某数是否存在于2.5亿数中）：先外部排序，2.5亿个整数可能有连续，构造数据结构 $[x, y]$ 表示从x到y之间的数均存在，再二分查找

## **5亿个int找它们的中位数**

---

这个例子比上面那个更明显。首先我们将int划分为 $2^{16}$ 个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到哪个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成 $2^{24}$ 个区域，然后确定区域的第几大数，在将该区域分成 $2^{20}$ 个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有 $2^{20}$ ，就可以直接利用direct addr table进行统计了。

# 集群

## 负载均衡

集群中的应用服务器（节点）通常被设计成无状态，用户可以请求任何一个节点。

负载均衡器会根据集群中每个节点的负载情况，将用户请求转发到合适的节点上。

负载均衡器可以用来实现高可用以及伸缩性：

- 高可用：当某个节点故障时，负载均衡器会将用户请求转发到另外的节点上，从而保证所有服务持续可用；
- 伸缩性：根据系统整体负载情况，可以很容易地添加或移除节点。

负载均衡器运行过程包含两个部分：

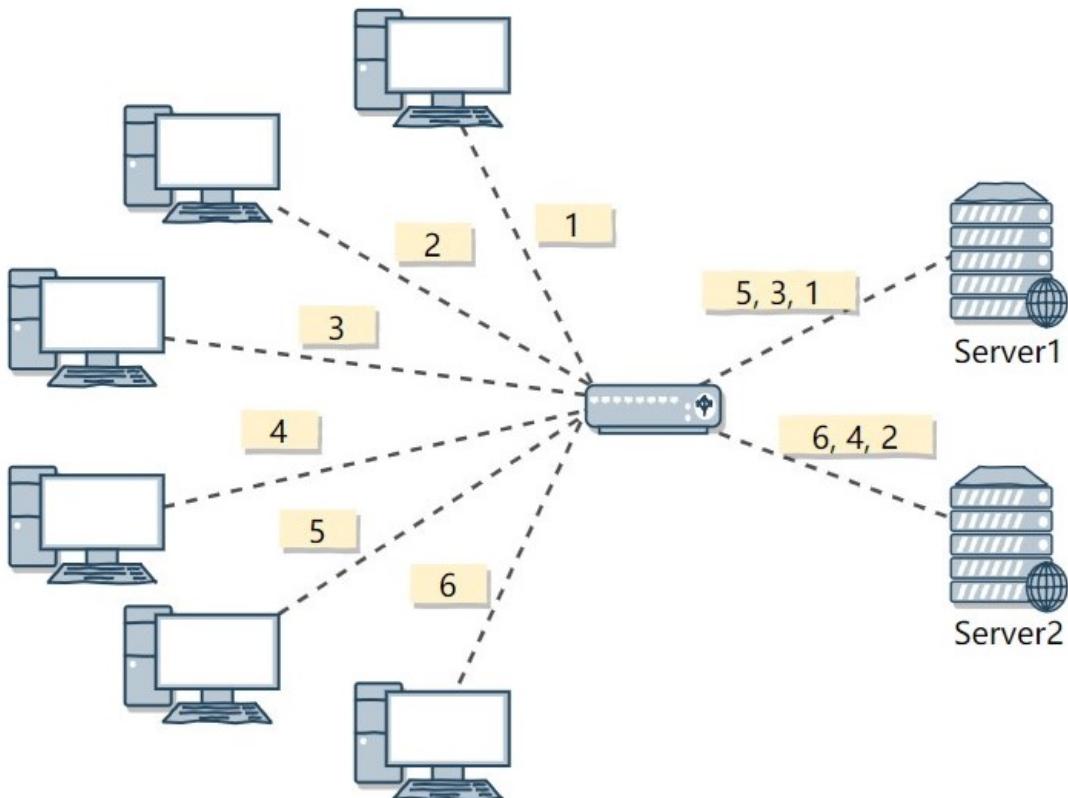
1. 根据负载均衡算法得到转发的节点；
2. 进行转发。

## 负载均衡算法

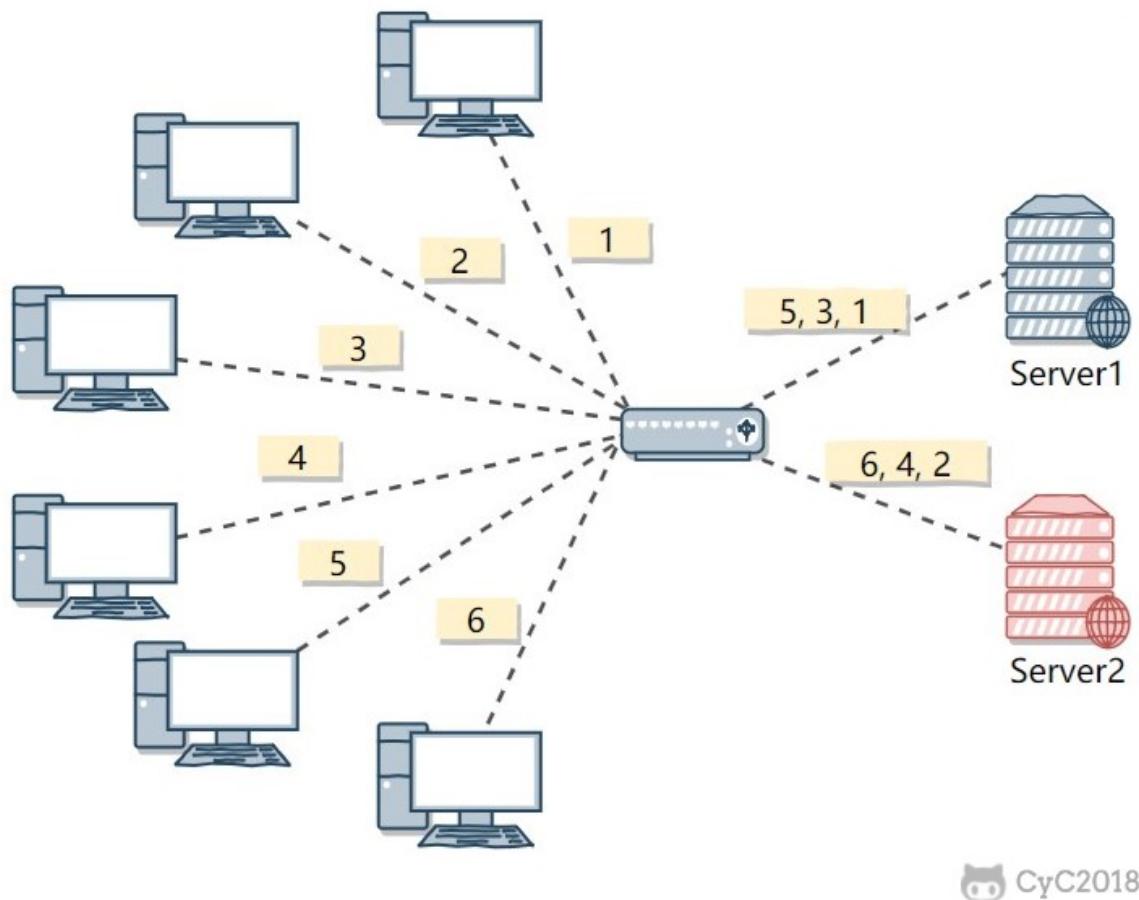
### 1. 轮询 (Round Robin)

轮询算法把每个请求轮流发送到每个服务器上。

下图中，一共有 6 个客户端产生了 6 个请求，这 6 个请求按 (1, 2, 3, 4, 5, 6) 的顺序发送。(1, 3, 5) 的请求会被发送到服务器 1，(2, 4, 6) 的请求会被发送到服务器 2。



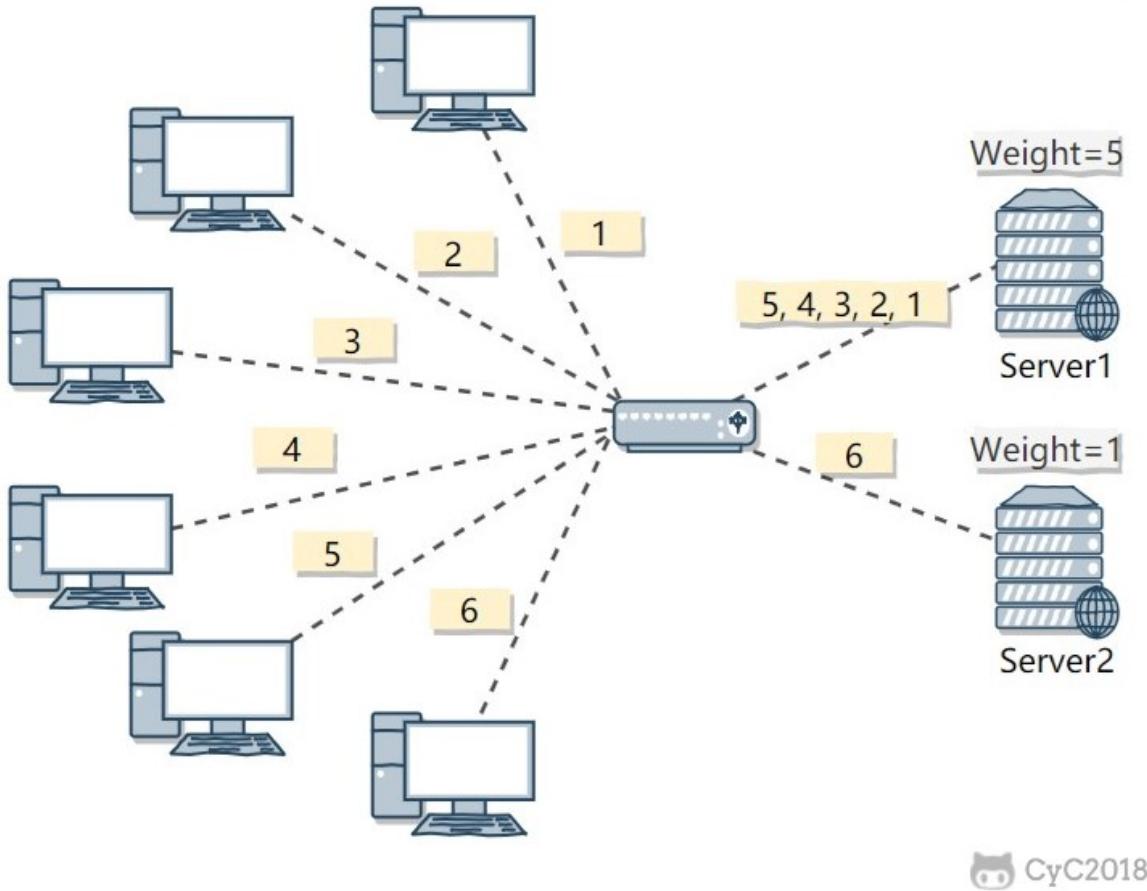
该算法比较适合每个服务器的性能差不多的场景，如果有性能存在差异的情况下，那么性能较差的服务器可能无法承担过大的负载（下图的 Server 2）。



## 2. 加权轮询 (Weighted Round Robin)

加权轮询是在轮询的基础上，根据服务器的性能差异，为服务器赋予一定的权值，性能高的服务器分配更高的权值。

例如上图中，服务器1被赋予的权值为5，服务器2被赋予的权值为1，那么(1, 2, 3, 4, 5)请求会被发送到服务器1，(6)请求会被发送到服务器2。

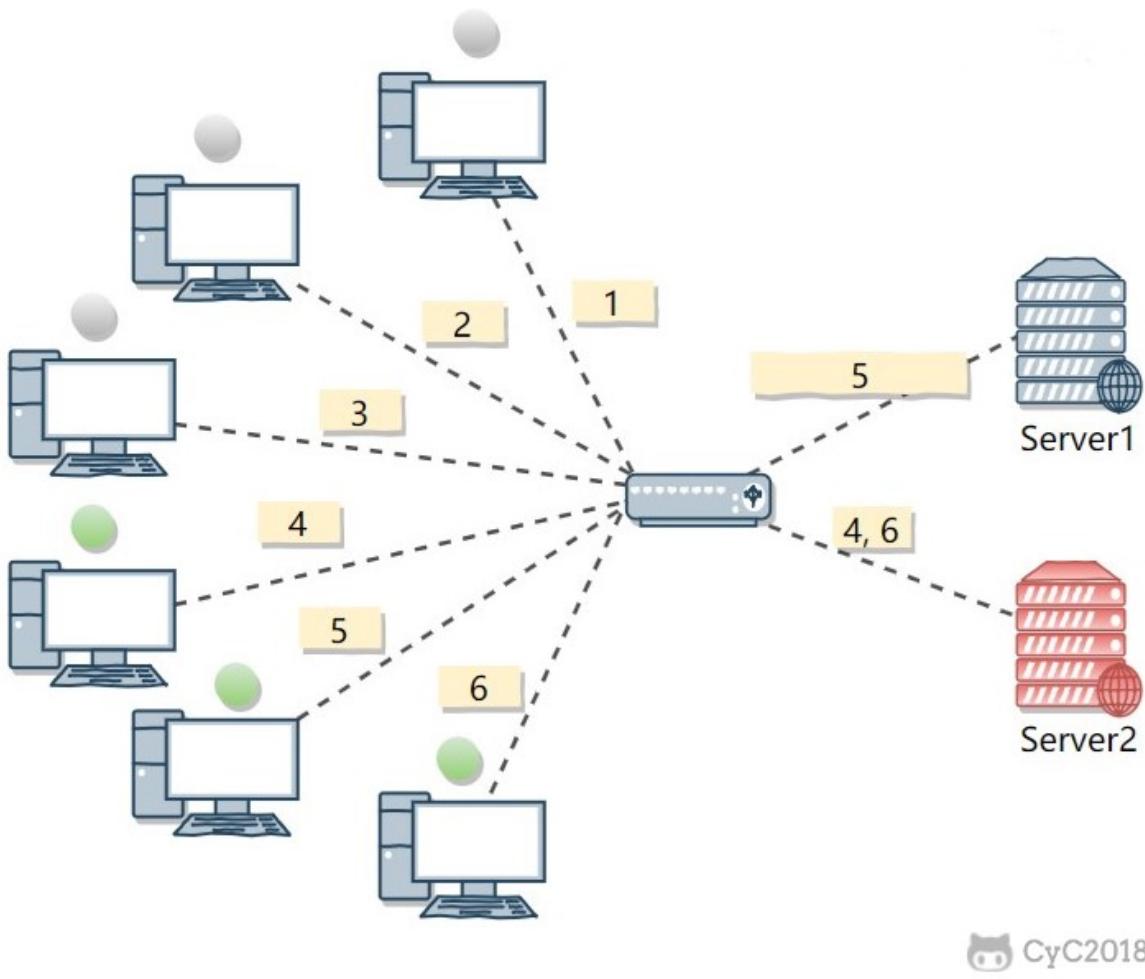


CyC2018

### 3. 最少连接 (least Connections)

由于每个请求的连接时间不一样，使用轮询或者加权轮询算法的话，可能会让一台服务器当前连接数过大，而另一台服务器的连接过小，造成负载不均衡。

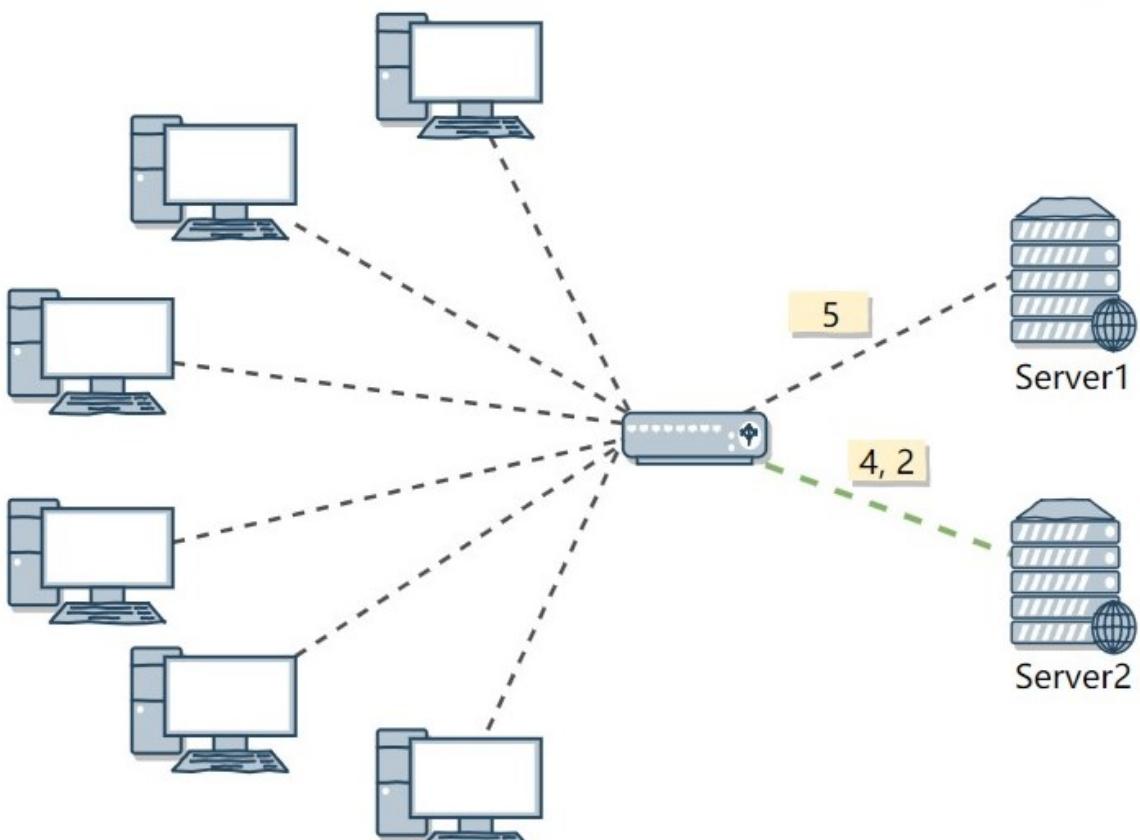
例如下图中，(1, 3, 5) 请求会被发送到服务器 1，但是 (1, 3) 很快就断开连接，此时只有 (5) 请求连接服务器 1；(2, 4, 6) 请求被发送到服务器 2，只有 (2) 的连接断开，此时 (6, 4) 请求连接服务器 2。该系统继续运行时，服务器 2 会承担过大的负载。



CyC2018

最少连接算法就是将请求发送给当前最少连接数的服务器上。

例如下图中，服务器 1 当前连接数最小，那么新到来的请求 6 就会被发送到服务器 1 上。



CyC2018

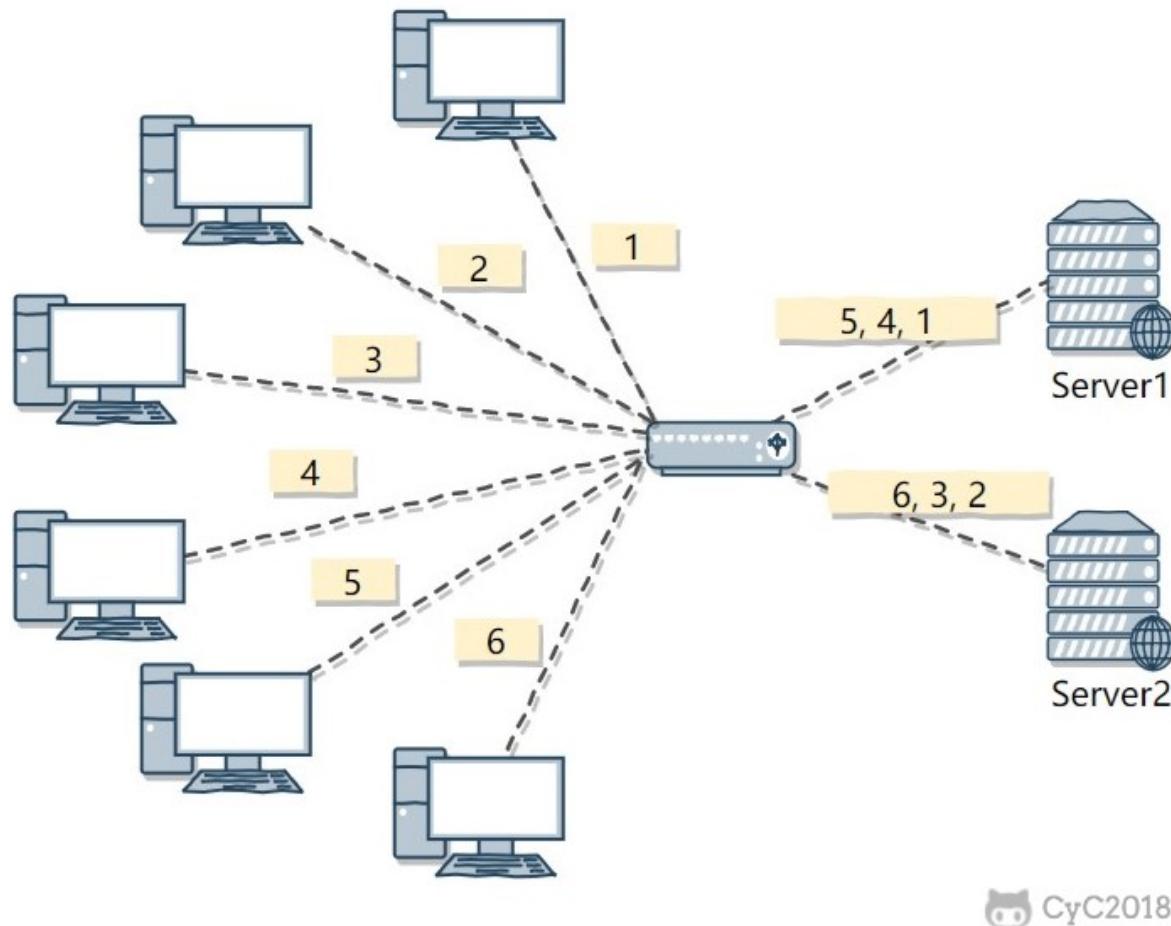
## 4. 加权最少连接 (Weighted Least Connection)

在最少连接的基础上，根据服务器的性能为每台服务器分配权重，再根据权重计算出每台服务器能处理的连接数。

## 5. 随机算法 (Random)

把请求随机发送到服务器上。

和轮询算法类似，该算法比较适合服务器性能差不多的场景。

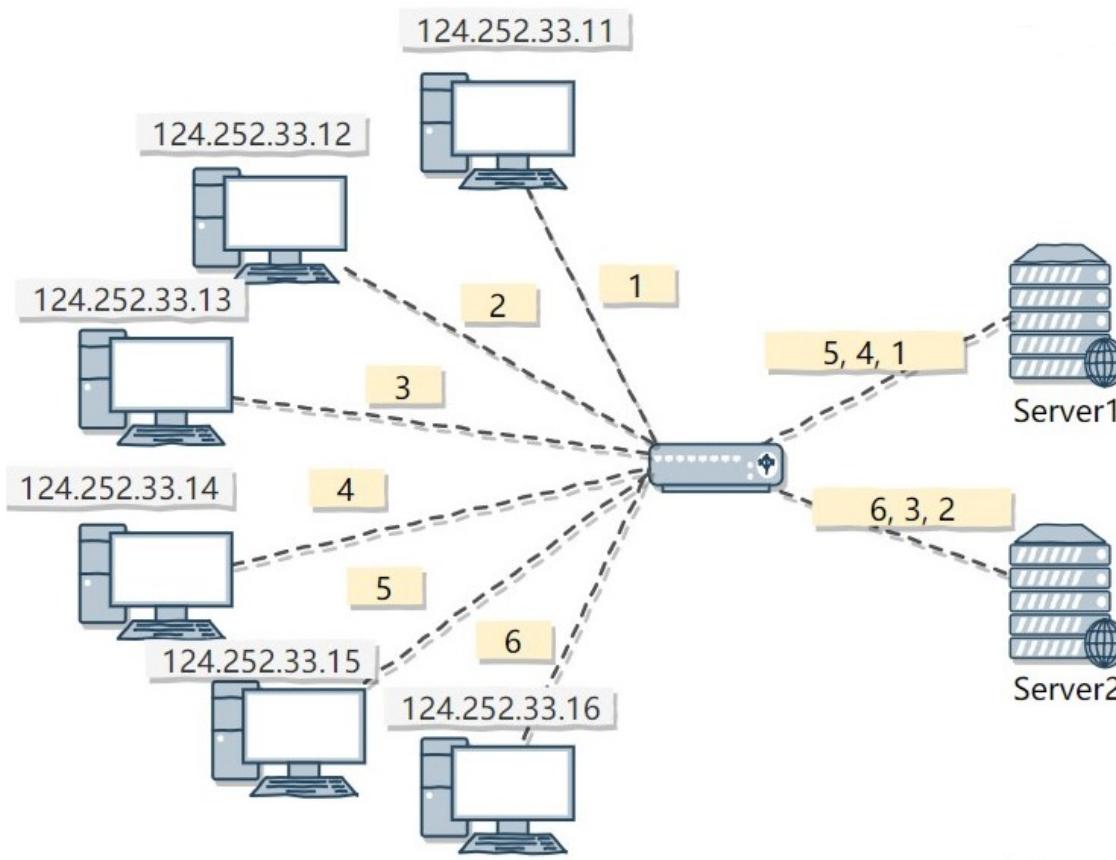


 CyC2018

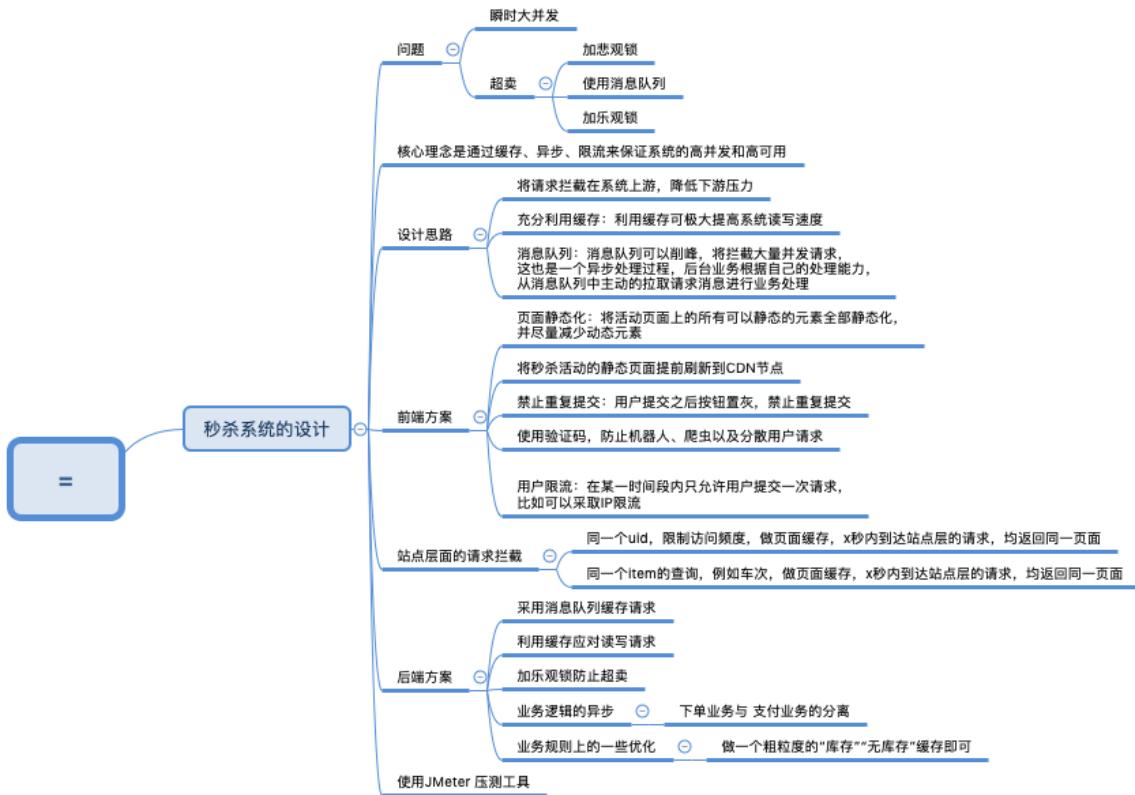
## 6. 源地址哈希法 (IP Hash)

源地址哈希通过对客户端 IP 计算哈希值之后，再对服务器数量取模得到目标服务器的序号。

可以保证同一 IP 的客户端的请求会转发到同一台服务器上，用来实现会话粘滞 (Sticky Session)



# 秒杀系统设计



## 解决思路

尽量将请求拦截在数据库的上游，因为一旦大量请求进入数据库，性能会急剧下降

## 架构和实现细节

- 前端模块（页面静态化、CDN、客户端缓存）
- 排队模块（Redis、队列实现异步下单）
- 服务模块（事务处理业务逻辑、避免并发问题）
- 防刷模块（验证码、限IP）

## 模块解析

### 前端模块

- 页面静态化，将后台渲染模板的方式改成使用HTML文件与AJAX异步请求的方式，减少服务端渲染开销，同时将秒杀页面提前放到CDN
- 客户端缓存，配置Cache-Control来让客户端缓存一定时间页面，提升用户体验
- 静态资源优化，CSS/JS/图片压缩，提升用户体验

### 排队模块

- 对Redis中的抢购对象预减库存，然后立即返回抢购成功请用户等待，这里利用了Redis将大部分请求拦截住，少部分流量进入下一阶段

- 如果参与秒杀的商品太多，进入下一阶段的流量依然比较大，则需要使用消息队列，Redis过滤之后的请求直接放入到消息队列，让消息队列进行流量的第二次削峰

## 服务模块

- 消息队列的消费者，业务逻辑是使用事务控制对数据库的下订单，减库存操作，且下订单操作要放到减库存操作之前，可以避免减库存update的行锁持有时间

## 防刷模块

- 针对恶意用户写脚本去刷，在Redis中保存用户IP与商品ID进行限制
- 针对普通用户疯狂的点击，使用JS控制抢购按钮，每几秒才能点击一次
- 在后台生成数学计算型的验证码，使用Graphics、BufferedImage实现图片，ScriptEngineManager计算表达式

## 异常流程的处理

- 如果在秒杀的过程中由于服务崩溃导致秒杀活动中断，那么没有好的办法，只能立即尝试恢复崩溃服务或者申请另寻时间重新进行秒杀活动
- 如果在下订单的过程中由于用户的某些限制导致下单失败，那么应该回滚事务，立即告诉用户失败原因

## 难点+坑+复盘优化

---

### 难点

- 理解整个架构设计的思路，围绕这个思路进行思考有什么方式可以做到，在开发过程中多进行压力测试反馈优化
- 代码中异常情况的处理与业务上应急预案的准备

### 坑

- 以上的解决方案能通过利用Redis与消息队列集群来承载非常高的并发量，但是运维成本高。比如Redis与消息队列都必须用到集群才能保证稳定性，会导致运维成本太高。所以需要有专业的运维团队维护。
- 避免同一用户同时下多个订单，需要写好业务逻辑或在订单表中加上用户ID与商品ID的唯一索引；避免卖超问题，在更新数量的sql上需要加上>0条件

### 优化

- 将7层负载均衡Nginx与4层负载均衡LVS一起使用进一步提高并发量
- 以上是应用架构上的优化，在部署的Redis、消息队列、数据库、虚拟机偏向选择带宽与硬盘读写速度高的
- 提前预热，将最新的静态资源同步更新到CDN的所有节点上，在Redis中提前加载好需要售卖的产品信息
- 使用分布式限流减少Redis访问压力，在Nginx中配置并发连接数与速度限制，但如果有很多不同的活动同时进行则不适用

# 消息队列

---

## 使用场景

- 异步处理

发送者将消息发送给消息队列之后，不需要同步等待消息接收者处理完毕，而是立即返回进行其它操作。消息接收者从消息队列中订阅消息之后异步处理。

例如在注册流程中通常需要发送验证邮件来确保注册用户身份的合法性，可以使用消息队列使发送验证邮件的操作异步处理，用户在填写完注册信息之后就可以完成注册，而将发送验证邮件这一消息发送到消息队列中。

只有在业务流程允许异步处理的情况下才能这么做，例如上面的注册流程中，如果要求用户对验证邮件进行点击之后才能完成注册的话，就不能再使用消息队列。

- 流量削峰

在高并发的场景下，如果短时间有大量的请求到达会压垮服务器。

可以将请求发送到消息队列中，服务器按照其处理能力从消息队列中订阅消息进行处理。

- 应用解耦

如果模块之间不直接进行调用，模块之间耦合度就会很低，那么修改一个模块或者新增一个模块对其它模块的影响会很小，从而实现可扩展性。

通过使用消息队列，一个模块只需要向消息队列中发送消息，其它模块可以选择性地从消息队列中订阅消息从而完成调用。

## 缺点

---

- 系统可用性降低

系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？如何保证消息队列的高可用，可以[点击这里查看](#)。

- 系统复杂度提高

硬生生加个 MQ 进来，你怎么[保证消息没有重复消费](#)？怎么[处理消息丢失的情况](#)？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

- 一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

## RocketMQ的一些问题

---

### 消费模式

Rocketmq消费分为push和pull两种方式，push为被动消费类型，pull为主动消费类型，push方式最终还是会从broker中pull消息。不同于pull的是，push首先要注册消费监听器，当监听器处触发后才开始消费消息，所以被称为“被动”消费。

### 消息丢失

当你系统需要保证百分百消息不丢失，你可以使用生产者每发送一个消息，Broker 同步返回一个消息发送成功的反馈消息。即每发送一个消息，同步落盘后才返回生产者消息发送成功，这样只要生产者得到了消息发送生成的返回，事后除了硬盘损坏，都可以保证不会消息丢失。

## 消息堆积

根据不同的业务实现不同的丢弃任务。

## 顺序消息

生产者生产消息时指定特定的 MessageQueue，消费者消费消息时，消费特定的 MessageQueue，其实单机版的消息中心在一个 MessageQueue 就天然支持了顺序消息。注意：同一个 MessageQueue 保证里面的消息是顺序消费的前提是：消费者是串行的消费该 MessageQueue，因为就算 MessageQueue 是顺序的，但是当并行消费时，还是会有顺序问题，但是串行消费也同时引入了两个问题：

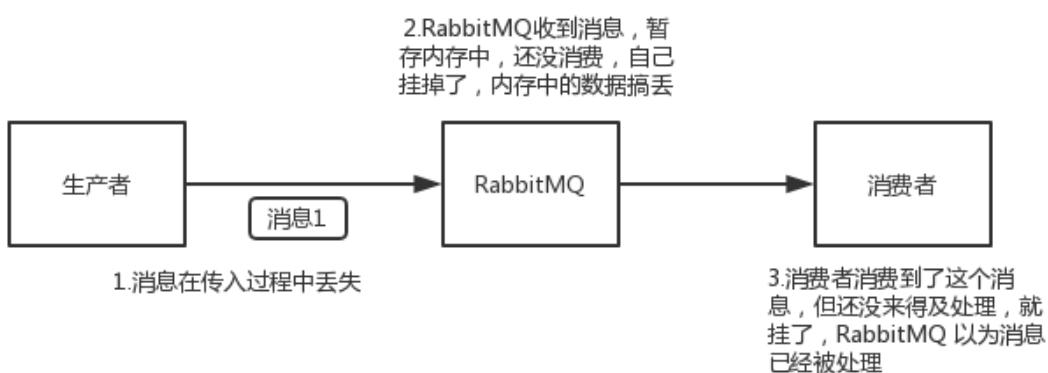
1. 引入锁来实现串行
2. 前一个消费阻塞时后面都会被阻塞

## 消息重复

- RocketMQ 会出现消息重复发送的问题，因为在网络延迟的情况下，这种问题不可避免的发生，如果非要实现消息不可重复发送，那基本太难，因为网络环境无法预知，还会使程序复杂度加大，因此默认允许消息重复发送
- RocketMQ 让使用者在消费者端去解决该问题，即需要消费者端在消费消息时支持幂等性的去消费消息
- 最简单的解决方案是每条消费记录有个消费状态字段，根据这个消费状态字段来是否消费或者使用一个集中式的表，来存储所有消息的消费状态，从而避免重复消费
- 具体实现可以查询关于消息幂等消费的解决方案

## 消息丢失

### RabbitMQ 消息丢失的 3 种情况



## 生产者弄丢了数据

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 `channel.txSelect`，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 `channel.txRollback`，然后重试发送消息；如果收到了消息，那么可以提交事务 `channel.txCommit`。

```
// 开启事务
channel.txSelect
try {
    // 这里发送消息
} catch (Exception e) {
    channel.txRollback

    // 这里再次重发这条消息
}

// 提交事务
channel.txCommit
```

但是问题是，RabbitMQ 事务机制（同步）一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 `confirm` 模式，在生产者那里设置开启 `confirm` 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 `ack` 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 `nack` 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 `confirm` 机制最大的不同在于，**事务机制是同步的**，你提交一个事务之后会阻塞在那儿，但是 `confirm` 机制是**异步的**，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你的一个接口通知你这个消息接收到了。

所以一般在生产者这块**避免数据丢失**，都是用 `confirm` 机制的。

## RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须**开启 RabbitMQ 的持久化**，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，**恢复之后会自动读取之前存储的数据**，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，**可能导致少量数据丢失**，但是这个概率较小。

设置持久化有**两个步骤**：

- 创建 queue 的时候将其设置为持久化  
这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是它是不会持久化 queue 里的数据的。
- 第二个是发送消息的时候将消息的 `deliveryMode` 设置为 2  
就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

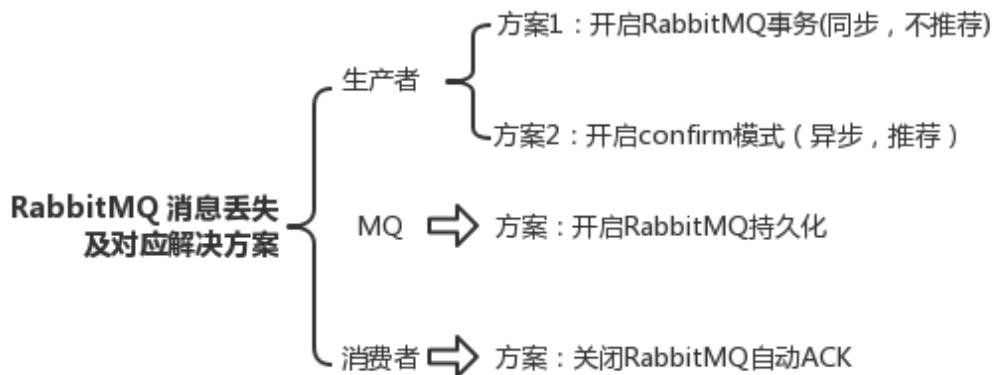
注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失。

所以，持久化可以跟生产者那边的 `confirm` 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 `ack` 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 `ack`，你也是可以自己重发的。

## 消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 `ack` 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 `ack`，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack` 了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



## 分布式消息的实现

1. 需要前置知识：2PC
2. RocketMQ4.3 起支持，原理为2PC，即两阶段提交，prepared->commit/rollback
3. 生产者发送事务消息，假设该事务消息 Topic 为 Topic1-Trans，Broker 得到后首先更改该消息的 Topic 为 Topic1-Prepared，该 Topic1-Prepared 对消费者不可见。然后定时回调生产者的本地事务A执行状态，根据本地事务A执行状态，来是否将该消息修改为 Topic1-Commit 或 Topic1-Rollback，消费者就可以正常找到该事务消息或者不执行等

注意，就算是事务消息最后回滚了也不会物理删除，只会逻辑删除该消息

## RocketMQ 不使用 ZooKeeper 作为注册中心的原因，以及自制的 NameServer 优缺点

- ZooKeeper 作为支持顺序一致性的中间件，在某些情况下，它为了满足一致性，会丢失一定时间内的可用性，RocketMQ 需要注册中心只是为了发现组件地址，在某些情况下，RocketMQ 的注册中心可以出现数据不一致性，这同时也是 NameServer 的缺点，因为 NameServer 集群间互不通信，它们之间的注册信息可能会不一致
- 另外，当有新的服务器加入时，NameServer 并不会立马通知到 Producer，而是由 Producer 定时去请求 NameServer 获取最新的 Broker/Consumer 信息（这种情况是通过 Producer 发送消息时，负载均衡解决）

**如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？**

**大量消息在 mq 里积压了几个小时了还没解决**

几千万条数据在 MQ 里积压了七八个小时，从下午 4 点多，积压到了晚上 11 点多。这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复 consumer 的问题，让它恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是 1000 条，一秒 3 个消费者是 3000 条，一分钟就是 18 万条。所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概 1 小时的时间才能恢复过来。

一般这个时候，只能临时紧急扩容了，具体操作步骤和思路如下：

- 先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。
- 新建一个 topic, partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。
- 然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，**消费之后不做耗时的处理**，直接均匀轮询写入临时建立好的 10 倍数量的 queue。
- 接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。
- 等快速消费完积压数据之后，**得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息**。

## mq 中的消息过期失效了

假设你用的是 RabbitMQ, RabbitMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是**大量的数据会直接搞丢**。

这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是**批量重导**，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上 12 点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。

假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

## mq 都快写满了

如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，**消费一个丢弃一个，都不要了**，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

## 如何保证RocketMQ高可用？

推荐的几种 Broker 集群部署方式，这里的 Slave 不可写，但可读，类似于 Mysql 主备方式。

- 单个 Master

这种方式风险较大，一旦 Broker 重启或者宕机时，会导致整个服务不可用，不建议线上环境使用。

- 多 Master 模式

一个集群无 Slave，全是 Master，例如 2 个 Master 或者 3 个 Master

优点：配置简单，单个 Master 宕机或重启维护对应用无影响，在磁盘配置为 RAID10 时，即使机器宕机不可恢复情况下，由于 RAID10 磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢）。性能最高。

缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到受到影响。

```
####先启动 NameServer  
####在机器 A, 启动第一个 Master  
####在机器 B, 启动第二个 Master
```

- 多 Master 多 Slave 模式, 异步复制

每个 Master 配置一个 Slave, 有多对Master-Slave, HA 采用异步复制方式, 主备有短暂消息延迟, 毫秒级。

优点: 即使磁盘损坏, 消息丢失的非常少, 且消息实时性不会受影响, 因为 Master 宕机后, 消费者仍然可以从 Slave 消费, 此过程对应用透明。不需要人工干预。性能同多 Master 模式几乎一样。

缺点: Master 宕机, 磁盘损坏情况, 会丢失少量消息。

```
####先启动 NameServer  
####在机器 A, 启动第一个 Master  
####在机器 B, 启动第二个 Master  
####在机器 C, 启动第一个 Slave  
####在机器 D, 启动第二个 Slave
```

- 多 Master 多 Slave 模式, 同步双写

每个 Master 配置一个 Slave, 有多对Master-Slave, HA 采用同步双写方式, 主备都写成功, 向应用返回成功。 优点: 数据与服务都无单点, Master宕机情况下, 消息无延迟, 服务可用性与 数据可用性都非常高

缺点: 性能比异步复制模式略低, 大约低 10%左右, 发送单个消息的 RT 会略高。目前主宕机后, 备机不能自动切换为主机, 后续会支持自动切换功能。

```
####先启动 NameServer  
####在机器 A, 启动第一个 Master  
####在机器 B, 启动第二个 Master  
####在机器 C, 启动第一个 Slave  
####在机器 D, 启动第二个 Slave
```

以上 Broker 与 Slave 配对是通过指定相同的brokerName 参数来配对, Master 的 BrokerId 必须是 0, Slave 的BrokerId 必须是大于 0 的数。另外一个 Master 下面可以挂载多个 Slave, 同一 Master 下的多个 Slave通过指定不同的 BrokerId 来区分。

## 如何设计消息队列

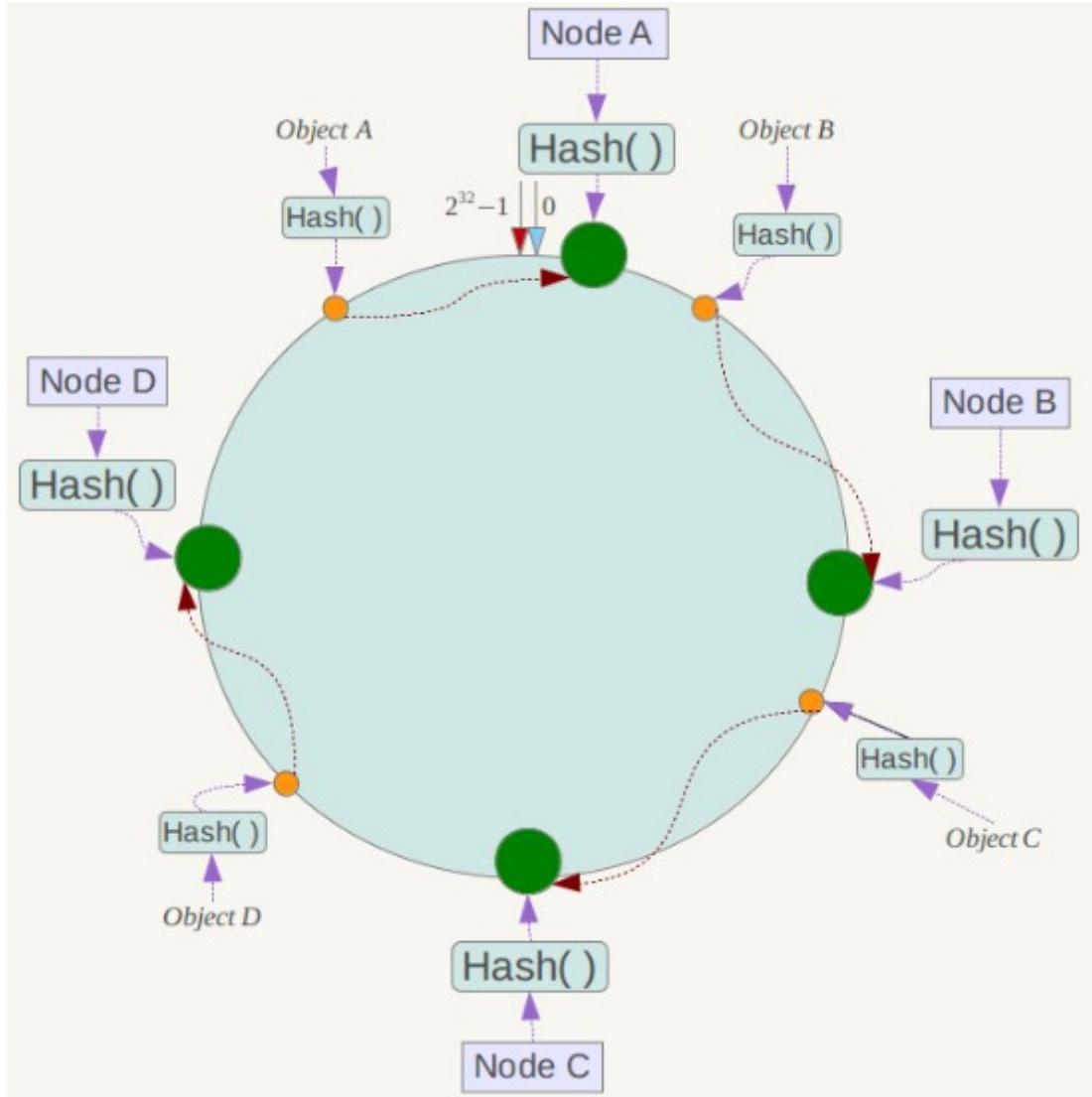
- 首先这个 mq 得支持可伸缩性吧, 就是需要的时候快速扩容, 就可以增加吞吐量和容量, 那怎么搞? 设计个分布式的系统呗, 参照一下 kafka 的设计理念, broker -> topic -> partition, 每个 partition 放一个机器, 就存一部分数据。如果现在资源不够了, 简单啊, 给 topic 增加 partition, 然后做数据迁移, 增加机器, 不就可以存放更多数据, 提供更高的吞吐量了?
- 其次你得考虑一下这个 mq 的数据要不要落地磁盘吧? 那肯定要了, 落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊? 顺序写, 这样就没有磁盘随机读写的寻址开销, 磁盘顺序读写的性能是很高的, 这就是 kafka 的思路。
- 其次你考虑一下你的 mq 的可用性啊? 这个事儿, 具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。
- 能不能支持数据 0 丢失啊? 可以的, 参考我们之前说的那个 kafka 数据零丢失方案。

# 一致性哈希

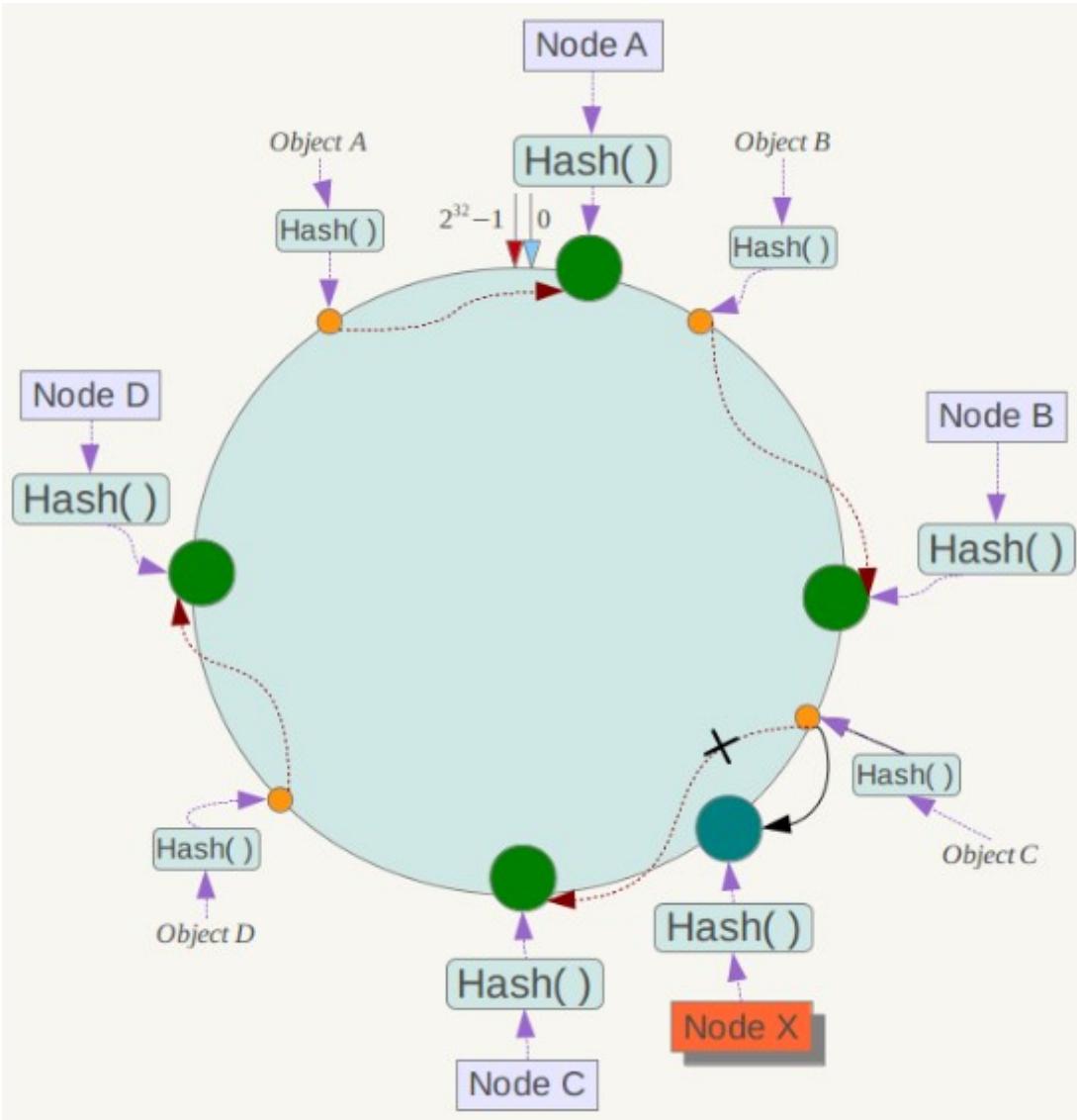
Distributed Hash Table (DHT) 是一种哈希分布方式，其目的是为了克服传统哈希分布在服务器节点数量变化时大量数据迁移的问题。

## 基本原理

将哈希空间  $[0, 2n-1]$  看成一个哈希环，每个服务器节点都配置到哈希环上。每个数据对象通过哈希取模得到哈希值之后，存放到哈希环中顺时针方向第一个大于等于该哈希值的节点上。



一致性哈希在增加或者删除节点时只会影响到哈希环中相邻的节点，例如下图中新增节点 X，只需要将它前一个节点 C 上的数据重新进行分布即可，对于节点 A、B、D 都没有影响。



## 虚拟节点

上面描述的一致性哈希存在数据分布不均匀的问题，节点存储的数据量有可能会存在很大的不同。

数据不均匀主要是因为节点在哈希环上分布的不均匀，这种情况在节点数量很少的情况下尤其明显。

解决方式是通过增加虚拟节点，然后将虚拟节点映射到真实节点上。虚拟节点的数量比真实节点来得多，那么虚拟节点在哈希环上分布的均匀性就会比原来的真实节点好，从而使得数据分布也更加均匀。