## 垃圾回收

1. JVM 内存区域

虚拟机栈、本地方法栈、程序计数器、本地内存、堆

1. 如何识别垃圾
   1. 引用计数法

不能解决循环引用

* 1. 可达性算法

GC Root 引用链，引用链外的是可回收对象

1. 垃圾回收主要方法
   1. 标记清除法

问题：内存碎片

* 1. 复制法

解决内存碎片，但是空间减少一半，移动存活对象效率低下

* 1. 标记整理法

清除垃圾后需要频繁移动存活对象，效率低下

* 1. 分代收集算法

综合以上算法优点

新生代 ：老年代 1 : 2

新生代 Eden : s0 : s1 8:1:1

出现原因：98%的对象生命很短，一次Minor GC就会被回收、根据对象存活周期不同，出现分代收集算法

新生代🡪 Young GC（Minor GC）老年代 Old GC（Full GC）

对象晋升到老年代：

1. 对象年龄达到阈值
2. 大对象
3. s0或s1相同年龄对象大小之和大于s0或s1空间一半以上，年龄大于等于该年龄的对象

把新生代设置成 Eden, S0，S1区或者给对象设置年龄阈值或者默认把新生代与老年代的空间大小设置成 1:2 都是为了**尽可能地避免对象过早地进入老年代，尽可能晚地触发 Full GC**。

1. 垃圾回收器对比
2. 在新生代工作的垃圾回收器：Serial, ParNew, ParallelScavenge
3. 在老年代工作的垃圾回收器：CMS，Serial Old, Parallel Old
4. 同时在新老生代工作的垃圾回收器：G1

Serial收集器：Serial 收集器是工作在新生代的，单线程的垃圾收集器

ParNew收集器：ParNew 收集器是 Serial 收集器的多线程版本，除了使用多线程，其

他像收集算法,STW,对象分配规则，回收策略与 Serial 收集器完成一样

CMS 收集器是以实现最短 STW 时间为目标的收集器，如果应用很重视服务的响应速度，希望给用户最好的体验，则 CMS 收集器是个很不错的选择！

## 2．Kafka

### 2.1 kafka基础

天然分布式消息队列

Producer –> Broker 🡪 Consumer

生产者和消费者就得知道：把数据丢给哪个队列，从哪个队列消费消息。我们需要给队列取名字，叫做**topic**(相当于数据库里边**表**的概念)

为了提高一个队列(topic)的**吞吐量**，Kafka会把topic进行分区(**Partition**)

一台Kafka服务器叫做**Broker**，Kafka集群就是多台Kafka服务器

一个topic会分为多个partition，实际上partition会**分布**在不同的broker中

**Kafka是天然分布式的**。

高可用：

Kafka是这样做的：我们数据存在不同的partition上，那kafka就把这些partition做**备份**。比如，现在我们有三个partition，分别存在三台broker上。每个partition都会备份，这些备份散落在**不同**的broker上。

**备份分区仅仅用作于备份，不做读写。**如果某个Broker挂了，那就会选举出其他Broker的partition来作为主分区，这就实现了**高可用**。

### 2.2 速度快的原因

1. 而partition持久化到磁盘是IO顺序访问的，并且是先写缓存，隔一段时间或者数据量足够大的时候才批量写入磁盘的

2. 消费者在读的时候也很有讲究：正常的读磁盘数据是需要将内核态数据拷贝到用户态的，而Kafka 通过调用sendfile()直接从内核空间（DMA的）到内核空间（Socket的），**少做了一步拷贝**的操作。

### 2.3 消费进度offset

Kafka就是用offset来表示消费者的消费进度到哪了，每个消费者会都有自己的offset。说白了offset就是表示消费者的**消费进度**。

### 2.4 zookeeper

既然提到了Zookeeper，那就多说一句。Zookeeper虽然在新版的Kafka中没有用作于保存客户端的offset，但是Zookeeper是Kafka一个重要的依赖。

* 探测broker和consumer的添加或移除。
* 负责维护所有partition的领导者/从属者关系（主分区和备份分区），如果主分区挂了，需要选举出备份分区作为主分区。
* 维护topic、partition等元配置信息
* ….

维持完全的顺序性在分布式系统看来几乎是无意义的。因为，如果需要维持顺序性，那么就只能有一条线程阻塞的处理顺序消息，即，Producer -> MQ -> Consumer 必须线程上一一对应。将有先后关系的消息发送到相同的分区上，即可解决大部分问题。

## 3. RabbitMQ

### 3.1基本架构

Producer 🡪 broker 🡪 Consumer

* Connection： 就是一个TCP的连接。Producer和Consumer都是通过TCP连接到RabbitMQ Server的。以后我们可以看到，程序的起始处就是建立这个TCP连接。
* Channels： 虚拟连接。它建立在上述的TCP连接中。数据流动都是在Channel中进行的。也就是说，一般情况是程序起始建立TCP连接，第二步就是建立这个Channel。

消息发送到Exchange 根据binding关系， exchange将消息发送到Queue

交换机类型：

1. Direct：消息中的路由键（routing key）如果和 Binding 中的 binding key 一致， 交换器就将消息发到对应的队列中。它是完全匹配、单播的模式。
2. Fanout: 每个发到 fanout 类型交换器的消息都会分到所有绑定的队列上去。fanout 交换器不处理路由键，只是简单的将队列绑定到交换器上，每个发送到交换器的消息都会被转发到与该交换器绑定的所有队列上。

使用过程一般就是先new 出一个Fanout类型的交换机，然后往这个交换机上绑定多个队列queue，不同的消费者各自监听不同的队列，这就实现了广播效果，因为同一个消息，会分发到所有队列中。

同一服务的多个实例都共享生产者的消息（多个实例都监听到广播），多实例间根据IP创建不同队列绑定带fanout交换机上。

1. Topic: topic 交换器通过模式匹配分配消息的路由键属性，将路由键和某个模式进行匹配，此时队列需要绑定到一个模式上。

### ACK机制

ACK 机制就是消费者从 RabbitMQ 收到消息并处理完成后，反馈给 RabbitMQ，然后 RabbitMQ 收到反馈后才将此消息从队列中删除。（先打上删除标记，之后再删除）

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。

这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑。

而且ack的机制可以起到限流的作用（Benefitto throttling）：在Consumer处理完成数据后发送ack，甚至在额外的延时后发送ack，将有效的balance Consumer的load

### 3．3持久化

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务

### 3.4 RPC

实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。

**RabbitMQ中实现RPC的机制是：**

* 客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了14中properties，这些属性会随着消息一起发送）中设置两个值replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）
* 服务器端收到消息并处理
* 服务器端处理完消息后，将生成一条应答消息到replyTo指定的Queue，同时带上correlationId属性
* 客户端之前已订阅replyTo指定的Queue，从中收到服务器的应答消息后，根据其中的correlationId属性分析哪条请求被执行了，根据执行结果进行后续业务处理

## Redis

### Zset

## Spring

### spring aop

1. **这样意味着Spring最终会将使用了@Aspect注解的类构建成Advisor对象后保存进BeanFactoryAspectJAdvisorsBuilder.advisorsCache中。**做了扫描@Aspect注解的扫描器是AbstractAutoProxyCreator类
2. 生成强化后的对象就藏在wrapIfNecessary中。在此处会从advisorsCache中根据aspectName取出对应的Advisor。拿到Advisor后，便是进行过滤的地方了，通过F8往后走，可以看到过滤的地方在AopUtils.canApply接口中。可以看到此处传进来的targetClass符合切面的要求，因此可以进行构建强化对象。在AbstractAutoProxyCreator的createProxy函数中看到，最后会构造出一个强化后的chatService。通过源码分析，我们发现在AbstractAutoProxyCreator构建强化对象的时候是调用了createAopProxy函数，重点来了，我们可以看到针对targetClass，也就是ChatService做了判断，如果targetClass有实现接口或者targetClass是Proxy的子类，那么使用的是JDK的动态代理实现AOP，如果不是才会使用CGLIB实现动态代理。

### 5.2 spring定时任务

### 5.3 Spring IOC

#### ****5.3.1传统开发模式的弊端****

1. 没有做到**资源复用、**许多组件只需要实例化一个对象就够了，创建多个没有任何意义。
2. **变化的代价太大** 假设有 10 个 Controller 依赖了 UserService，最开始实例化的是 UserServiceImpl，后面需要换一个实现类 OtherUserServiceImpl，我就得逐个修改那 10 个 Controller，非常麻烦。更换实现类的需求可能不会太多，没多大说服力。该数据源组件要想真正生效需要对其进行许多配置，这个创建和配置过程是非常麻烦的。而且配置可能会随着业务需求的变化经常更改，这时候你就需要修改每一个依赖该组件的地方，牵一发而动全身。这还只是演示了一个数据源的创建配置过程，真实开发中可有太多组件和太多配置需要编码了，其麻烦程度堪称恐怖。

创建了许多重复对象，造成大量资源浪费；

更换实现类需要改动多个地方；

创建和配置组件工作繁杂，给组件调用方带来极大不便。

**组件的调用方参与了组件的创建和配置工作**

其实，调用方只需关注组件如何调用，至于这个组件如何创建和配置又与调用方有什么关系呢？如果我们编码时，有一个「东西」能帮助我们创建和配置好那些组件，我们只负责调用该多好。这个「东西」就是容器。

#### ****5.3.2控制反转和依赖注入****

控制反转，是指对象的创建和配置的控制权从调用方转移给容器。有了 IoC 容器，我们可以将对象交由容器管理，交由容器管理后的对象称之为 Bean。调用方不再负责组件的创建，要使用组件时直接获取 Bean 即可。调用方只需按照约定声明依赖项，所需要的 Bean 就自动配置完毕了，就好像在调用方外部注入了一个依赖项给其使用，所以这种方式称之为 依赖注入（Dependency Injection，缩写为 DI）。控制反转和依赖注入是一体两面，都是同一种开发模式的表现形式。

IoC 轻而易举地解决了我们刚刚总结的问题：

对象交由容器管理后，默认是单例的，这就解决了资源浪费问题。

若要更换实现类，只需更改 Bean 的声明配置，即可达到无感知更换：

现在组件的使用和组件的创建与配置完全分离开来。调用方只需调用组件而无需关心其他工作，这极大提高了我们的开发效率，也让整个应用充满了灵活性、扩展性。

### 5.4 spring异步 @Async

spring 在扫描bean的时候会扫描方法上是否包含@Async注解，如果包含，spring会为这个bean动态地生成一个子类（即代理类，proxy），代理类是继承原来那个bean的。此时，当这个有注解的方法被调用的时候，实际上是由代理类来调用的，代理类在调用时增加异步作用。然而，如果这个有注解的方法是被同一个类中的其他方法调用的，那么该方法的调用并没有通过代理类，而是直接通过原来的那个bean，所以就没有增加异步作用，我们看到的现象就是@Async注解无效。

## Caffeine

### 6.1 缓存策略 ****W-TinyLFU****

一般比较简单的缓存就会直接用到LFU(Least Frequently Used，即最不经常使用)或者LRU(Least Recently Used，即最近最少使用)

## 集合框架

### HashMap

**数组和链表组合构成**的数据结构。

数组里面每个地方都存了Key-Value这样的实例，在Java7叫Entry在Java8中叫Node。

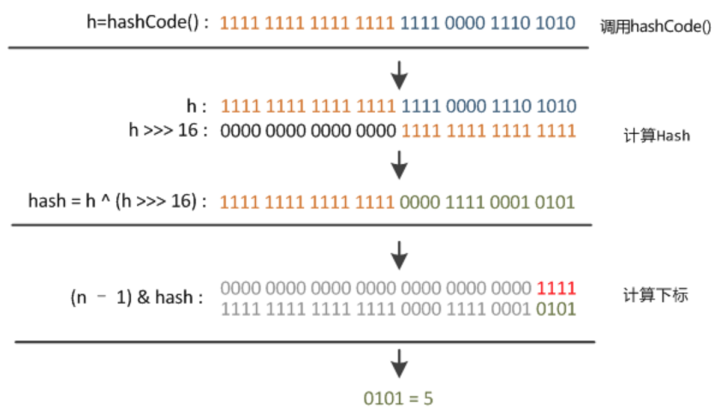
链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。

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**。

因为位运算直接对内存数据进行操作，不需要转成十进制，所以位运算要比取模运算的效率更高，所以HashMap在计算元素要存放在数组中的index的时候，使用位运算代替了取模运算。之所以可以做等价代替，前提是要求HashMap的容量一定要是2^n 。

**指定容量初始化**

当我们通过HashMap(int initialCapacity)设置初始容量的时候，HashMap并不一定会直接采用我们传入的数值，而是经过计算，得到一个新值，目的是提高hash的效率。(1->1、3->4、7->8、9->16) 把一个数转化成第一个比他自身大的2的幂



**java8之前是头插法**，就是说新来的值会取代原有的值，原有的值就顺推到链表中去，就像上面的例子一样，因为写这个代码的作者认为后来的值被查找的可能性更大一点，提升查找的效率。**在java8之后，都是所用尾部插入了。**

HashMap的扩容机制：

数组容量是有限的，数据多次插入的，到达一定的数量就会进行扩容，也就是resize。

有两个因素：

Capacity：HashMap当前长度。

LoadFactor：负载因子，默认值0.75f。

就比如当前的容量大小为100，当你存进第76个的时候，判断发现需要进行resize了，那就进行扩容

分为两步

扩容：创建一个新的Entry空数组，长度是原数组的2倍。

ReHash：遍历原Entry数组，把所有的Entry重新Hash到新数组。

**快速失败**

**快速失败（fail—fast）**是java集合中的一种机制， 在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception

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。

### ConcurrentHashMap

JDK1.7---由 Segment 数组、HashEntry 组成，和 HashMap 一样，仍然是**数组加链表**。原理上来说，ConcurrentHashMap 采用了**分段锁**技术，其中 Segment 继承于 ReentrantLock。

JDK 1.8 抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。

### ArrayList

ArrayList就是数组列表，主要用来装载数据，当我们装载的是基本类型的数据int，long，boolean，short，byte…的时候我们只能存储他们对应的包装类，它的主要底层实现是数组Object[] elementData。

ArrayList可以通过构造方法在初始化的时候指定底层数组的大小。

通过无参构造方法的方式ArrayList()初始化，则赋值底层数Object[] elementData为一个默认空数组Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {}所以数组容量为0，只有真正对数据进行添加add时，才分配默认DEFAULT\_CAPACITY = 10的初始容量。

l**ArrayList的大小是如何自动增加的**

当有人试图在arraylist中增加一个对象的时候，Java会去检查arraylist，以确保已存在的数组中有足够的容量来存储这个新的对象。如果没有足够容量的话，那么就会新建一个长度更长的数组，旧的数组就会使用Arrays.copyOf方法被复制到新的数组中去，现有的数组引用指向了新的数组。

**在索引中ArrayList的增加或者删除某个对象的运行过程？效率很低吗？解释一下为什么？**

在ArrayList中增加或者是删除元素，要调用System.arraycopy这种效率很低的操作，如果遇到了需要频繁插入或者是删除的时候，你可以选择其他的Java集合，比如LinkedList。

**当传递ArrayList到某个方法中，或者某个方法返回ArrayList，什么时候要考虑安全隐患？如何修复安全违规这个问题呢？**

当array被当做参数传递到某个方法中，如果array在没有被复制的情况下直接被分配给了成员变量，那么就可能发生这种情况，即当原始的数组被调用的方法改变的时候，传递到这个方法中的数组也会改变。

### 7.4 BlockingQueue

BlockingQueue 是一个**先进先出**的队列（Queue），为什么说是阻塞（Blocking）的呢？是因为 BlockingQueue 支持当获取队列元素但是队列为空时，会阻塞等待队列中有元素再返回；也支持添加元素时，如果队列已满，那么等到队列可以放入新元素时再放入。

## 并发

### 8.1 ****synchronized****

synchronized是java提供的原子性内置锁，这种内置的并且使用者看不到的锁也被称为**监视器锁**，使用synchronized之后，会在编译之后在同步的代码块前后加上monitorenter和monitorexit字节码指令，他依赖操作系统底层互斥锁实现。他的作用主要就是实现原子性操作和解决共享变量的内存可见性问题。

### 8.2 ****volatile原理****

相比synchronized的加锁方式来解决共享变量的内存可见性问题，volatile就是更轻量的选择，他没有上下文切换的额外开销成本。使用volatile声明的变量，可以确保值被更新的时候对其他线程立刻可见。volatile使用内存屏障来保证不会发生指令重排，解决了内存可见性的问题。

### 8.3. start()

请求JVM运行一个新的线程；

让两个线程同时运行： new Thread().start()

子线程 准备工作，就绪 上下文、PC

不能重复执行一个线程的start方法

### 8.4 ThreadLocal

只能被一个线程访问

Thread 类中 ThreadLocalMap可以保存本线程的所有threadLocal变量

ThreadLocal两种应用：

* + 1. 最开始调用用户实现的initValue方法初始化；
    2. 直接调用set方法设置值。

调用ThreadLocal.get()方法 ，如果获取到的值为空，会触发initValue方法初始化

ThreadLocalMap的entry是弱引用

### 8.5. 线程池

#### 8.5.1 线程池中的线程执行时异常

到此线程池中的线程异常了会被怎么处理讲完了，总结一下就是：

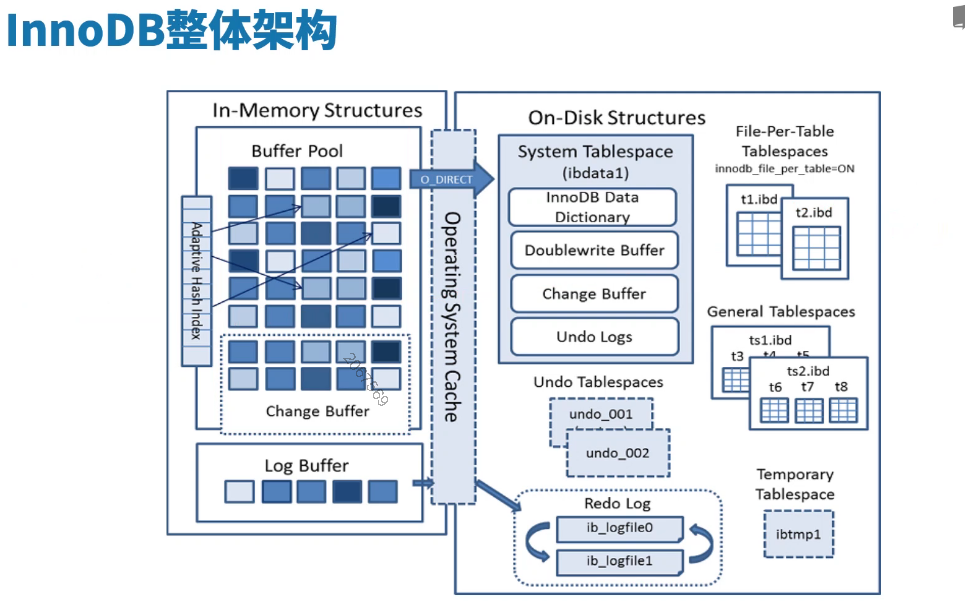
1、execute方法,可以看异常输出在控制台，而submit在控制台没有直接输出，必须调用Future.get()方法时，可以捕获到异常。

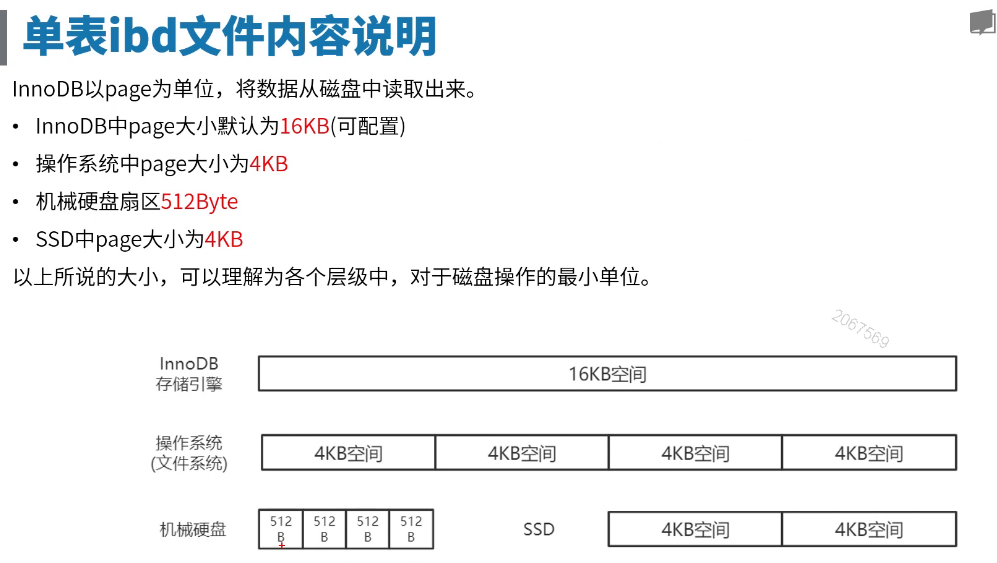
2、一个线程出现异常不会影响线程池里面其他线程的正常执行。

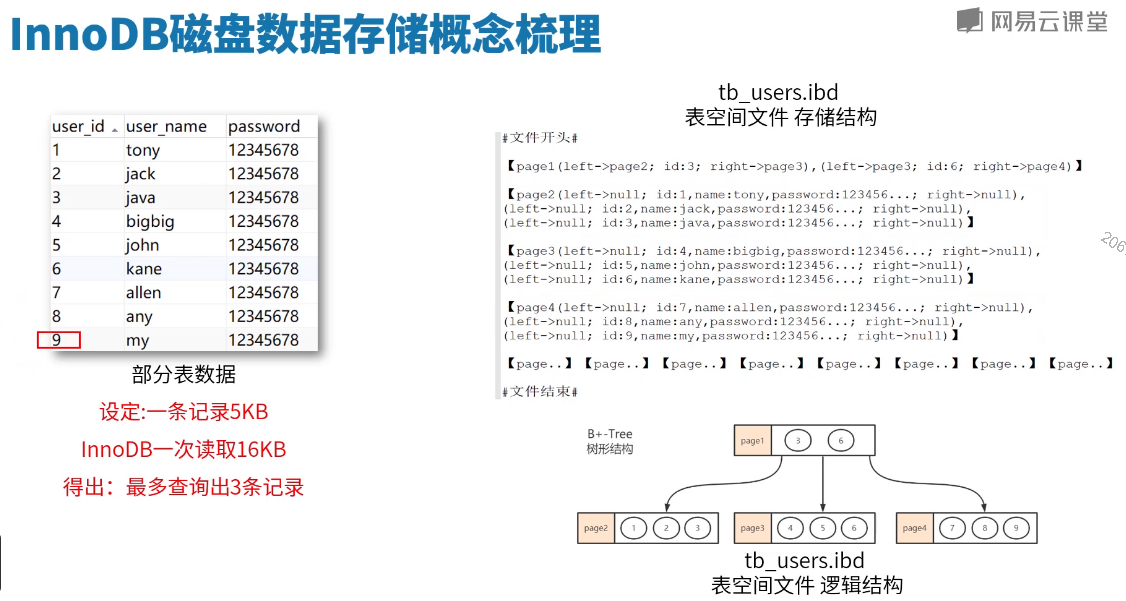
3、线程不是被回收而是线程池把这个线程移除掉，同时创建一个新的线程放到线程池中。

## Mysql

### 9.2 InnoDB







### 索引

Hash、B+

哈希表的特点就是**可以快速的精确查询，但是不支持范围查询**。

等值查询的场景，就只有KV（Key，Value）的情况

**有序数组**，它就比较优秀了呀，它在等值查询的和范围查询的时候都很Nice。有序的适合静态数据，因为如果我们新增、删除、修改数据的时候就会改变他的结构。可以用来做静态存储引擎啊，用来保存静态数据，例如你2019年的支付宝账单，2019年的淘宝购物记录等等都是很合适的，都是不会变动的历史数据。

二叉树是有序的，所以是支持范围查询的。

但是他的时间复杂度是O(log(N))，为了维持这个时间复杂度，更新的时间复杂度也得是O(log(N))，那就得保持这棵树是完全平衡二叉树了。

B树的表示要比完全平衡二叉树要“矮”，原因在于B树中的一个节点可以存储多个元素。

B树其实就已经是一个不错的数据结构，用来做索引效果还是不错的

B+树的表示要比B树要“胖”，原因在于B+树中的非叶子节点会冗余一份在叶子节点中，并且叶子节点之间用指针相连。

索引优化：

* 1. Like 走索引，如果起始字段确定；
  2. 索引列为空 可以走索引

1. 在开发涉及SQL的业务都会去本地环境跑一遍SQL，用explain去看一下执行计划，看看分析的结果是否符合自己的预期，用没用到相关的索引
2. 缓存的问题，我们在执行SQL的时候，记得加上SQL NoCache去跑SQL，这样跑出来的时间就是真实的查询时间了。
3. 最开始提到了用执行计划去分析，我想explain是大家SQL调优都会回答到的吧
4. 需要回表这样的操作，那我们怎么能做到不回表呢？在自己的索引上就查到自己想要的，不要去主键索引查了。

覆盖索引

如果在我们建立的索引上就已经有我们需要的字段，就不需要回表了

联合索引

建立联合索引包含要查询的信息

1. 最做匹配原则
2. 索引下推 MySQL 5.6 引入的索引下推优化（index condition pushdown)， 可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

### 9.2 InnoDB索引

#### 主键索引（聚簇索引）

每个InnoDB表都有一个聚簇索引 ，聚簇索引使用B+树构建，叶子节点存储的数据是整行记录

#### 组合索引

**最左匹配原则：**

最左前缀匹配原则和联合索引的**索引存储结构和检索方式**是有关系的。

在组合索引树中，最底层的叶子节点按照第一列a列从左到右递增排列，但是b列和c列是无序的，b列只有在a列值相等的情况下小范围内递增有序，而c列只能在a，b两列相等的情况下小范围内递增有序。

可以说创建的idx\_abc(a,b,c)索引，相当于创建了(a)、（a,b）（a,b,c）三个索引。

**组合索引的最左前缀匹配原则：使用组合索引查询时，mysql会一直向右匹配直至遇到范围查询(>、<、between、like)就停止匹配。**

#### 避免回表

**使用覆盖索引的优化策略减少回表的情况。**

#### 联合索引的使用

**联合索引**，在建立索引的时候，尽量在多个单列索引上判断下是否可以使用联合索引。联合索引的使用不仅可以节省空间，还可以更容易的使用到索引覆盖。

**联合索引的创建原则**，在创建联合索引的时候因该把频繁使用的列、区分度高的列放在前面，也可以在常需要作为查询返回的字段上增加到联合索引中

### 9.3 事务

#### 9.3.1 事务的四大特性：ACID

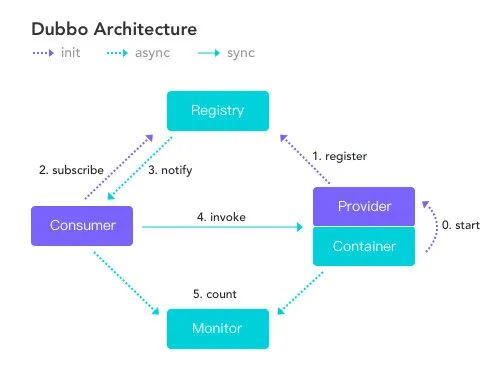
* 原子性
* 一致性
* 隔离性
* 持久性

#### 9.3.2 并发事务带来的问题：

1. 脏读 T1事务修改数据，被T2事务读取到
2. 丢失修改 多事务间修改覆盖
3. 不可重复读 T1事务两次读取之间，T2事务对相同数据进行修改
4. 幻读 T1事务两次读取之间，T2事务新增或删除数据

#### 9.3.3 事务的隔离级别

## Dubbo



| **节点** | **角色说明** |
| --- | --- |
| Consumer | 需要调用远程服务的服务消费方 |
| Registry | 注册中心 |
| Provider | 服务提供方 |
| Container | 服务运行的容器 |
| Monitor | 监控中心 |

## 服务消费者

消费者面向接口编程

公用jar包维护接口

框架提供接口的代理类，需要给代理调用方法与参数

代理从注册中心知晓调用哪个服务提供方

提供方集群部署、通过负载均衡来调用

容错机制

约定传输协议

约定序列化格式

中间加入filter，做统计

## 服务提供者

实现对应接口

暴露服务、向注册中心注册自己

和消费者协商协议来处理请求，做反序列化

反序列化后在线程池中处理，结果原路返回

## 注册中心

注册服务、动态变更通知订阅者

## 监控运维