Citi bank

1128, Thomas Router, Max
1207 RBC. Justin
1211, Quick Play , Fuat,
1212 LobLaw, Zach
Quick Play
fdf

reverse tinkliest, merge sort ,quick sort,
TODO code coutDown， singleton, Enum, thread sequence.? HashMap, equals, dashcode?
long resume, 80 hours
Aug mid, 14,

TODO function programming, thread test 50 interview question.
Spring knowledge. summerize.

online test. 1424 , rest API, Spring. 1415,

take a picture and send by to me.

Java knowledge

1,thread.join, will ensure, current thread end, then execute next thread.start()
executorServer submit will return Future.
java.lang.Thread.yield() method causes the currently executing thread object to temporarily pause and allow other threads to execute

3. hash map, put will cause resize(), at that moment, get() can cause infinite.

4. What happens if two different objects have the same hashcode?
They will just be added to the same bucket and equals() will be used to distinguish them. Each bucket can contain a list of objects with the same hash code. In theory you can return the same integer as a hash code for any object of given class, but that would mean that you loose all performance benefits of the hash map and, in effect, will store objects in a list
keys along with their associative values are stored in a linked list node in the bucket and the keys are essentially compared in hashmap using equals() method not by hashcode
When you lookup a value in the hashmap, by giving it a key, it will first look at the hash code of the key that you gave. The hashmap will then look into the corresponding bucket, and then it will compare the key that you gave with the keys of all pairs in the bucket, by comparing them with equals()

5. subString() memory leak
Substring creates a new object out of source string by taking a portion of original string
it calls String (int offset, int count, char value []) constructor to create new String object. What is interesting here is, value[], which is the same character array used to represent original string

5. java pass by reference or value

1)swap(int a, int b)->doen't change,
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、n
um2 的副本，副本的内容无论怎么修改，都不会影响到原件本身
2)
public static void change(int[] array) {
// 将数组的第一个元素变为 0
array[0] = 0;
}
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
3)
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。

6. Singtон thread safe
public class Singleton {
//必须使用 volatile,因为指令重拍，可能导致 instance is not null, but boject is not initialized. (1,3,2 step)
1。 memory =allocate
2. intance = init
3. instance = memory,

private static volatile Singleton _instance;

/**
* Double checked locking code on Singleton
* @return Singelton instance
*/
public static Singleton getInstance() {
if (_instance == null) {
synchronized (Singleton.class) {

```
if (_instance == null) {
_instance = new Singleton();
}
}
}
return _instance;
}

}
```

7. UncaughtExceptionHandler

```
Thread t = new Thread(new MyRunnable());
t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {

public void uncaughtException(Thread t, Throwable e) {
LOGGER.error(t + " throws exception: " + e);
}
});
t.start();
```

8. Abstract Factory and responsible for the creation of different hierarchies of objects based on the type of factory
9 override equal, especially if you want to do equality check based upon business logic rather than object equality

10. Is it better to synchronize critical sections of getInstance() method or the whole getInstance() method
because if we lock the whole method, then every time some some one call this method, it will have to wait even though we are not creating any object. In other words, synchronization is only needed, when you create object, which happens only once

11. Where does equals() and hashCode() method comes in the picture during the get() operation? (answer)
12. This core Java interview question is a follow-up of previous Java question and the candidate should know that once you mention hashCode, people are most likely ask, how they are used in HashMap. When you provide a key object, first it's hashcode method is called to calculate bucket location. Since a bucket may contain more than one entry as linked list, each of those Map.Entry object is evaluated by using equals() method to see if they contain the actual key object or not

13 What will be the problem if you don't override hashCode() method
wo objects which are equal by equals() must have the same hashcode. In this case, an other object may return different hashCode and will be stored on that location, which breaks invariant of HashMap class, because they are not supposed to allow duplicate keys

14 deadlock avoid: If lock will be acquired in a consistent order and released in just opposite order, there would not be a situation where one thread is holding a lock which is acquired by other and vice-versa

15. What is the difference between creating String as new() and literal
16. does not put the object str in String pool, we need to call String.intern() method which is used to put them into String pool explicitly. It's only when you create String object as a String literal e.g. String s = "Test" that Java automatically puts that into the String pool

17 Mostly Immutable classes are also final in Java, in order to prevent sub classes from overriding methods, which can compromise Immutability. You can achieve the same functionality by making member as non-final but private and not modifying them except in constructor

18
Java 7 adds a new exception class called ReflectiveOperationException. The Javadoc documentation describes this class as a "Common superclass of exceptions thrown by reflective operations in core reflection.

19
a reifiable type is one whose runtime representation contains same information than its compile-time representa-tion

a non-reifiable type is one whose runtime representation contains less information than its compile-time representa-tion
Arrays are reifiable as arrays remains as it is at runtime

20 enum and nested enum - package local, final and static

21 heap pollution is a situation that arises when a variable of a parameterized type refers to an object that is not of that parameterized type.[1] This situation is normally detected during compilation and indicated with an unchecked warning

22
you can make SimpleDateFormat thread-safe using ThreadLocal you achieve thread-safety without paying cost of synchronization or immutability

thread local need to implement initialValue() and get().
23. wait notify()
If you don't call them from synchronized context, your code will
throw IllegalMonitorStateException

24
A bucket is used to store key value pairs . A bucket can have multiple key-value pairs

hashvalue is used to find the bucket location at which the Entry object is stored . Entry object stores in the bucket like this (hash,key,value,bucketindex)
What if when two different keys have the same hashcode
Solution, equals() method comes to rescue So we traverse through linked list , comparing keys in each entries using keys.equals() until it return true
What if when two keys are same and have the same hashcode ?
If key needs to be inserted and already inserted hashkey's hashcodes are same, and keys are also same(via reference or using equals() method) then override the previous key value pair with the current key value pai

25
When we create a string in java like String s1="hello"; then an object will be created in string pool(hello) and s1 will be pointing to hello.Now if again we do String s2="hello"; then another object will not be created but s2 will point to hello because JVM will first check if the same object is present in string pool or not

26
What is the difference between creating String as new() and literal?
When we create string with new() Operator, it's created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

String s = new String("Test");

does not put the object in String pool , we need to call String.intern() method which is used to put them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool

27
A Vector defaults to doubling size of its array . While when you insert an element into the ArrayList , it increases
its Array size by 50% .

By default ArrayList size is 10 . It checks whether it reaches the last element then it will create the new array ,copy the new data of last array to new array ,then old array is garbage collected by the Java Virtual Machine (JVM) .

28
What is difference between Executor.submit() and Executer.execute() method ?
There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit any thrown exception, checked exception or

not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will re-throw this exception, wrapped in an ExecutionException

29 volatile, Thus when Thread A writes to a volatile variable V, and afterwards Thread B reads from variable V, any variable values that were visible to A at the time V was written are guaranteed now to be visible to B

30
The iterator in the HashMap is fail-safe (If you change the map while iterating, you'll know). The enumerator for the Hashtable is not fail-safe.

31，

B 树， 每个节点包括 M / 2 到 M 个，最多 M 个字节点， M 是树高， 叶结点在一层。
key 不重复， 父节点的关键码是子结点的分界， M =3, key = 1, or 2,
2-3 树， 子结点 2 个或者 2 个，

32

left = left+ (right-left)/2 , avoid overflow

33
string-> hashcode->array_index
two string can have the same hashcode , and array Index smaller than hashcode, so two hashCode can mapped to the same index, So 'chaining' can resolve this issue, if collision happened, store them to linkList.
oo(1) query time , o(n) for worst hash table.
question : how to resolve collision, , grow/shrink?

which means not only another thread has visibility of latest value of volatile variable but also all the variable is seen by the thread which has updated value of volatile variable before these threads sees it. Busy spin is one of the technique to wait for events without releasing CPU 1) Immutable objects are by default thread safe, can be shared without synchronization in concurrent environment.
2) Immutable object simplifies development, because its easier to share between multiple threads without external synchronization.

34. Another important benefit of Immutable objects is reusability String immutable
1. 1) Pool can be share
2. 3) Since String is immutable it can safely share between many threads which is very important for multithreaded

3. 4) Another reason of Why String is immutable in Java is to allow String to cache its hashcode, being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in hashmap in Java

35.

floating point calculation is pretty clear that result of floating point calculation may not be exact at all time and it should not be used in places where exact result is expected use double, 2.15 and 1.0 using double is: 1.0499999999999998, if bidDicimal, is stable.

System.out.println(3*0.1); can be 0.30000000000000004

byte：8 位，最大存储数据量是 255，存放的数据范围是-128~127 之间。

short：16 位，最大数据存储量是 65536，数据范围是-32768~32767 之间。

int：32 位，最大数据存储容量是 2 的 32 次方减 1，数据范围是负的 2 的 31 次方到正的 2 的 31 次方减 1。

long：64 位，最大数据存储容量是 2 的 64 次方减 1，数据范围为负的 2 的 63 次方到正的 2 的 63 次方减 1。

float：32 位，数据范围在 3.4e-45~1.4e38，直接赋值时必须在数字后加上 f 或 F。

double：64 位，数据范围在 4.9e-324~1.8e308，赋值时可以加 d 或 D 也可以不加。

boolean：只有 true 和 false 两个取值。

char：16 位，存储 Unicode 码，用单引号赋值。

36.

Java byte to string: String str = new String(bytes, "UTF-8");

37.

Can we cast an int value into byte variable? what will happen if the value of int is larger than byte? Yes, we can cast but int is 32 bit long in java while byte is 8 bit long in java so when you cast an int to byte higher 24 bits are lost and a byte can only hold a value from -128 to 128

38.

advice 类型：

* 前置通知(before advice)
* 返回后通知(after returning advice)
* 抛出异常后通知(after throwing advice)
* 后通知(after advice)
* 环绕通知(around advice)

* 连接点(joinpoint)：被拦截的方法

* 切入点(pointcut):对连接点进行拦截的定义。

39.

The += operator implicitly cast the result of addition into the type of variable used to hold the result

byte a = 127; byte b = 127; b = a + b; // error : cannot convert from int to byte b += a; // ok

40.
An Integer object will take more memory an Integer is the an object and it store meta data overhead about the object and int is primitive type so its takes less space

41.
Yes from Java 7 onward we can use String in switch case but it is just syntactic sugar. Internally string hash code is used for the switch

42.
Objects are created on the heap in Java irrespective of their scope e.g. local or member variable. while it's worth noting that class variables or static members are created in method area of Java memory space and both heap and method area is shared between different thread

+UseConcMarkSweepGC Concurrent Mark Sweep Garbage collector is most widely used garbage collector in java and it uses an algorithm to first mark object which needs to collect when garbage collection
Concurrent low pause Collector: This Collector is used if the -Xingc or -XX:+UseConcMarkSweepGC is passed on the command line. This is also referred as Concurrent Mark Sweep Garbage collector. The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection

Throughput Garbage Collector: This garbage collector in Java uses a parallel version of the young generation collector. It is used if the -XX:+UseParallelGC

-XX:+UseParallelGC should not be used with -XX:+UseConcMarkSweepGC
XX:+TraceClassLoading and -XX:+TraceClassUnloading print information class loads and unloads

43,
Counter prime = new Counter(); // prime holds a strong reference - line 2 SoftReference<Counter> soft = new SoftReference<Counter>(prime) ; //soft reference variable has SoftReference to Counter Object created at line 2 prime = null; // now Counter object is eligible for garbage collection but only be collected when JVM absolutely needs memory

Counter counter = new Counter(); // strong reference - line 1 WeakReference<Counter> weakCounter = new WeakReference<Counter>(counter); //weak reference counter = null; // now Counter object is eligible for garbage collection

44.
What is -XX:+UseCompressedOops JVM option? Why use it? (answer)

When you go migrate your Java application from 32-bit to 64-bit JVM, the heap requirement suddenly increases, almost double, due to increasing size of ordinary object pointer from 32 bit to 64 bit. This also adversely affect how much data you can keep in CPU cache, which is much smaller than memory. Since main motivation for moving to 64-bit JVM is to specify large heap size, you can save some memory by using compressed OOP. By using -XX:+UseCompressedOops, JVM uses 32-bit OOP instead of 64-bit OOP.

45.
System.out.println("JVM Bit size: " + System.getProperty("sun.arch.data.model"));

Output:
JVM Bit size: 32 //JVM is 32 bit
JVM Bit size: amd64 //JVM is 64 bit

46.

JRE stands for Java run-time and it's required to run Java application. JDK stands for Java development kit and provides tools to develop Java program e.g. Java compiler. It also contains JRE. The JVM stands for Java virtual machine and it's the process responsible for running Java application. The JIT stands for Just In Time compilation and helps to boost the performance of Java application by converting Java byte code into native code when the crossed certain threshold i.e. mainly hot code is converted into native code
JIT compile 技术，将运行频率很高的字节码直接编译为机器指令执行以提高性能，所以当字节码被 JIT 编译为机器码的时候，要说它是编译执行的也可以。也就是说，运行时，部分代码可能由 JIT 翻译为目标机器指令（以 method 为翻译单位，还会保存起来，第二次执行就不用翻译了）直接执行

47
The stack is used to hold method frames and local variables while objects are always allocated memory from the heap. The stack is usually much smaller than heap memory and also didn't shared between multiple threads, but heap is shared among all threads in JVM

48.
public static final variables are also known as a compile time constant, the public is optional there. They are replaced with actual values at compile time because compiler know their value up-front and also knows that it cannot be changed during run-time. One of the problem with this is that if you happened to use a public static final variable from some in-house or third party library and their value changed later than your client will still be using old value even after you deploy a new version of JARs

比如运行的时候替换另一个 jar

49.

Set is an unordered collection, you get no guarantee on which order element will be stored. Though some of the Set implementation e.g. LinkedHashSet maintains order.
Also SortedSet and SortedMap e.g. TreeSet and TreeMap maintains a sorting order, imposed by using Comparator or Comparable.

50
Both poll() and remove() take out the object from the Queue but if poll() fails then it returns null but if remove fails it throws Exception.

51
PriorityQueue 是个基于优先级堆的极大优先级队列。
此队列按照在构造时所指定的顺序对元素排序，既可以根据元素的自然顺序来指定排序（参阅 Comparable），也可以根据 Comparator 来指定
PriorityQueue guarantees that lowest or highest priority element always remain at the head of the queue, but LinkedHashMap maintains the order on which elements are inserted. When you iterate over a PriorityQueue, iterator doesn't guarantee any order but iterator of LinkedHashMap does guarantee the order on which elements are inserted

52
Comparator and sort all the elements of Collection on order defined by Comparator.If we don't pass any Comparator than object will be sorted based upon there natural order like String will be sorted alphabetically

53
The Comparable interface is used to define the natural order of object while Comparator is used to define custom order. Comparable can be always one, but we can have multiple comparators to define customized order for objects.

54
differences between direct and non-direct ByteBuffer derived from the fact that one is inside heap memory while other is outside heap.

ByteBuffer not only allows you to operate on heap byte arrays but also with direct memory, which resides outside the JVM

55.
volatile variable also follows happens-before relationship, which means any write happens before any read in volatile variable

56
çMain difference between FileInputStream and FileReader is that former is used to read binary data while later is used to read text data

57
you cannot override static method in Java because they are resolved at compile time rather than runtime

58
1. SOAP web services have all the advantages that web services has, some of the additional advantages are:
* WSDL document provides contract and technical details of the web services for client applications without exposing the underlying implementation technologies.
* SOAP uses XML data for payload as well as contract, so it can be easily read by any technology.
* SOAP protocol is universally accepted, so it's an industry standard approach with many easily available open source implementations.
2.
3. What are disadvantages of SOAP Web Services? Some of the disadvantages of SOAP protocol are:
* Only XML can be used, JSON and other lightweight formats are not supported.
* SOAP is based on the contract, so there is a tight coupling between client and server applications.
* SOAP is slow because payload is large for a simple string message, since it uses XML format.
* Anytime there is change in the server side contract, client stub classes need to be generated again.
* Can't be tested easily in browser
Caching refers to storing server response in client itself so that a client needs not to make server request for same resource again and again. A server response should have information about how a caching is to be done so that a client caches response for a period of time or never caches the server response

59 异常处理： http://blog.csdn.net/mahoking/article/details/45064259

60 stream:
Metaspace（JEP 122）代替持久代（PermGen space）

```
new Task( Status.OPEN, 5 ),
new Task( Status.OPEN, 13 ),
new Task( Status.CLOSED, 8 )

final long totalPointsOfOpenTasks = tasks
.stream()
.filter( task -> task.getStatus() == Status.OPEN )
.mapToInt( Task::getPoints )
.sum();


// Group tasks by their status
final Map< Status, List< Task > > map = tasks
.stream()
```

```
.collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );


Stream<Integer> intStream = Stream.of(1,2,3,4);
List<Integer> intList = intStream.collect(Collectors.toList());
```

Intermediate Operations
filter

```
tream<String> names = Stream.of("aBc", "d", "ef");
System.out.println(names.map(s -> {
return s.toUpperCase();
}).collect(Collectors.toList()));
```

findFirst() return optional.

stateless lambda expressions: If you are using parallel stream and lambda expressions are stateful, it can result in random responses.

61
http://www.baeldung.com/spring-boot-actuators

62.
volatile is slow from 355, to 1000 ms. to print 1M

63 Oath2
用户访问 web 游戏应用，该游戏应用要求用户通过 Facebook 登录。用户登录到了 Facebook,再重定向会游戏应用， 游戏应用就可以访问用户在 Facebook 的数据了，并且该应用可以代表用户向 Facebook 调用函数(如发送状态更新)。

64
```
@RequestMapping(value = "/media/v1/domaincategories/{domainName}", produces =
MediaType.APPLICATION_JSON_UTF8_VALUE, method = RequestMethod.GET)
@Transactional  public ResponseEntity<String> domainCategories(final @RequestHeader
MultiValueMap<String,String> headers,   final @PathVariable("domainName") String
domainName, final @RequestParam(value = "localeId", required = false)
```

```
// Code to swap 'x' and 'y'
x = x + y;  // x now becomes 15
y = x - y;  // y becomes 10
x = x - y;  // x becomes 5
```

65 binary search 注意是小于等于

```
while (first <= last) {   int middle = (first + last) / 2;   if (arr[middle].flavor < search) {   first = middle + 1;   } else if (arr[middle].flavor > search) {   last = middle - 1;   } else if (arr[middle].flavor == search) {   return arr[middle].index;   }  }
```

1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

Implement LRU cache

What are different states in lifecycle of Thread?
When we create a Thread in java program, its state is New. Then we start the thread that change it’s state to Runnable. Thread Scheduler is responsible to allocate CPU to threads in Runnable thread pool and change their state to Running. Other Thread states are Waiting, Blocked and Dead. Read this post t

B treee,

ki ki+1, if <ki, then go left,
m 阶 b 树， 每个节点关键码个数位 3/2-》（1，2）。
插入时候要保证 关键码还是， 1，2， 这样就可能把节点上移， 通常吧中间的节点上移动
删除的时候，如果关键码数量不够了， 可以从左右兄弟分

Chef,
define ‘recipe’ default.rb -> imageMagic install
ELB-> instance node.

Spring cloud config:
publish property from GIT to web service
config-client-dev.properties
in client
we can use blew setting, then can read value from file directly
@Value("${foo}")

spring.application.name=config-client spring.cloud.config.label=master
spring.cloud.config.profile=dev

zuul:
routes:
api-a:
path: /api-a/**
serviceId: service-ribbon
api-b:
path: /api-b/**
serviceId: service-feign

Ribbon
通过@EnableDiscoveryClient 向服务中心注册；并且向程序的 ioc 注入一个 bean: restTemplate;并通过@LoadBalanced 注解表明这个 restRemplate 开启负载均衡的功能
* 一个服务注册中心，eureka server,端口为 8761
* service-hi 工程跑了两个实例，端口分别为 8762,8763，分别向服务注册中心注册
* sercvice-ribbon 端口为 8764,向服务注册中心注册
* 当 sercvice-ribbon 通过 restTemplate 调用 service-hi 的 hi 接口时，因为用 ribbon 进行了负载均衡，会轮流的调用 service-hi：8762 和 8763 两个端口的 hi 接口；

touch Dockerfile
docker build -t cs-human-task-service .

docker images 以验证是否已正确创建映像。
Copy
docker images --filter reference=hello-world

docker login 命令。此命令提供一个在 12 小时内有效的授权令牌

Elastic load balancing: You have the option to use a load balancer with your service. Amazon ECS can create an Elastic Load Balancing (ELB) load balancer to distribute the traffic across the container instances your task is launched on.
*

Spring interview

Advice: Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application.

We can also use @Component, @Service, @Repository and @Controller annotations with classes to configure them to be as spring bean

@Autowired annotation with fields or methods for autowiring byType
this annotation to work, we also need to enable annotation based configuration in spring bean configuration file. This can be done by context:annotation-config element.

Controller class is responsible to handle different kind of client requests based on the request mappings. We can create a controller class by using @Controller annotation. Usually it's used with @RequestMapping annotation to define handler methods for specific URI mapping.

@Component is used to indicate that a class is a component. These classes are used for auto detection and configured as bean, when annotation based configurations are used.

@Controller is a specific type of component, used in MVC applications and mostly used with RequestMapping annotation.
@Repository annotation is used to indicate that a component is used as repository and a mechanism to store/retrieve/search data. We can apply this annotation with DAO pattern implementation classes.
@Service is used to indicate that a class is a Service. Usually the business facade classes that provide some services are annotated with this.

@ComponentScan 注解可以从当前 package 和子 package 中自动加载其他的 Spring 组件
@EnableAutoConfiguration，告诉 Spring Boot 采用自动配置。

@EnableWebSecurity.

http.csrf().disable();
configure(HttpSecurity http)
antMatchers(HttpMethod.POST, MediaServiceUrl.MEDIA_STATUS.getUrl()).authenticated().
Divide spring bean configurations based on their concerns such as spring-jdbc.xml, spring-security.xml.

f you are using Aspects, make sure to keep the join pint as narrow as possible to avoid advice on unwanted methods. Consider custom annotations that are easier to use and avo

1. For smaller applications, annotations are useful but for larger applications annotations can become a pain. If we have all the configuration in xml files, maintaining it will be easier.

+docker run -v $(pwd)/docker_share:/secret -e APP_NAME=$APP_NAME -e EXPEDIA_ENVIRONMENT=$EXPEDIA_ENVIRONMENT -e AWS_REGION=$AWS_REGION -e "ACTIVE_VERSION=$(git rev-parse HEAD)" -p 8080:8080 $(docker images -q | head -1)

Ansible: based on SSH, Python

ansible-playbook -i inventory/victor -u valarcon -kK stopAll.yml
playbook contains some task

inventory-server name:
group-vars that is like db url, user name
role: which mean different compoment
role/meta/install software that is used by this component
role/task/ task include some action , like cp configuration file to /cof, lib, setup log
we can do deploy/stop/restart

exition elliot building

type erasure means that generic type information is not available to the JVM at runtime, only compile time.
The reasoning behind major implementation choice is simple – preserving backward compatibility with older versions of Java. When a generic code is compiled into bytecode, it will be as if the generic type never existed. This me

Generic
List<? extends Number> list = new ArrayList<Number>(); //Upper bounder
list.add(new Integer(3)); //compile error

Number num = list.get(0);, //finde, when number OUT, it is fine

2. Lower bound

List<? super Number> list = new ArrayList<Number>(); //lower bounder
list.add(new Integer(3)); // fine, add(data in), works fine
Number num = list.get(0);, //error, when number OUT,

* 如果参数化类型表示一个 T 的生产者，使用 < ? extends T>;
* 如果它表示一个 T 的消费者，就使用 < ? super T>;

java8
Runnable r1 = () -> System.out.println("My Runnable");

We need Default Methods because of the following reasons:
* It allow us to provide method's implementation in Interfaces.
* To add new Functionality to Interface without breaking the Classes which implement that Interface.
* java.util.Date is Mutable and not Thread-Safe.
* java.text.SimpleDateFormat is not Thread-Safe.
* Less Performance.
Java SE 8's Date and Time API has the following Advantages compare to Java's OLD Date API.
* Very simple to use.
* Human Readable Syntax that is More Readability.
* All API is Thread-Safe.

Java8 有一个新特性叫做 Default Methods，可以在接口中加入方法体。
们称之为 default method，现在可以加入到接口提供默认的方法实现。举例：

Diamond Problem in Inheritance

```
public interface A{
default void display() { //code goes here }
}
public interface B extends A{ }
public interface C extends A{ }
public class D implements B,C{ }
```

Java 7 have issue
java 8

```
public interface A{
default void display() { //code goes here }
}
public interface B extends A{ }
public interface C extends A{ }
public class D implements B,C{
void display() {
B.super.display();
}
}
```

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it's execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads.

What is context-switching

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time.

How can we make sure main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function

Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That's why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

How can we achieve thread safety in Java?

There are several ways to achieve thread safety in java – synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword, using immutable classes and Thread safe classes

Which is more preferred – Synchronized method or Synchronized block?

Synchronized block is more preferred way because it doesn't lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.

What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it's variables, so if the variable is not thread safe, we can use synchronization but if we

want to avoid synchronization, we can use ThreadLocal variables. Every thread has it's own ThreadLocal variable and they can use it's get() and set() methods to get the default value or change it's value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread

```
private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
@Override
protected SimpleDateFormat initialValue()
{
return new SimpleDateFormat("yyyyMMdd HHmm");
}
};
```

```
t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){

@Override
public void uncaughtException(Thread t, Throwable e) {
System.out.println("exception occured:"+e.getMessage());
}
```

java time task interface, schedule a time

```
TimerTask timerTask = new MyTimerTask();
//running timer task as daemon thread
Timer timer = new Timer(true);
timer.scheduleAtFixedRate(timerTask, 0, 10*1000);
```

ThreadPool example:

```
ExecutorService executor = Executors.newFixedThreadPool(5);
for (int i = 0; i < 10; i++) {
Runnable worker = new WorkerThread("" + i);
executor.execute(worker);
}
executor.shutdown();
```

```
//SchedulePool
ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
//schedule to run after sometime
System.out.println("Current Time = "+new Date());
for(int i=0; i<3; i++){
Thread.sleep(1000);
WorkerThread worker = new WorkerThread("do heavy processing");
scheduledThreadPool.schedule(worker, 10, TimeUnit.SECONDS);
```

Atomic operations are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

Lock
* it's possible to make them fair
* it's possible to make a thread responsive to interruption while waiting on a Lock object.
* it's possible to try to acquire the lock, but return immediately or after a timeout if the lock can't be acquired
* it's possible to acquire and release locks in different scopes, and in different orders

To Avoid ConcurrentModificationException in multi-threaded environment
1. You can convert the list to an array and then iterate on the array. This approach works well for small or medium size list but if the list is large then it will affect the performance a lot.
2. You can lock the list while iterating by putting it in a synchronized block. This approach is not recommended because it will cease the benefits of multithreading.
3. If you are using JDK1.5 or higher then you can
use ConcurrentHashMap and CopyOnWriteArrayList classes. This is the recommended approach to avoid concurrent modification exceptio

Java Stack memory is used for execution of a thread. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

ClassNotFoundException is a checked exception which occurs when an application tries to load a class through its fully-qualified name and can not find its definition on the classpath.
This occurs mainly when trying to load classes
using Class.forName(), ClassLoader.loadClass() or ClassLoader.findSystemClass(). Therefore, we need to be extra careful of

NoClassDefFoundError
NoClassDefFoundError is a fatal error. It occurs when JVM can not find the definition of the class while trying to:
* Instantiate a class by using the new keyword
* Load a class with a method call
The error occurs when a compiler could successfully compile the class, but Java runtime could not locate the class file. It usually happens when there is an exception while executing a static block or initializing static fields of the class, so class initialization fails

HankerRank SQL:

```
SELECT
COUNT(CITY) - COUNT(DISTINCT CITY)
FROM
STATION;
```

```
select city, length(city) from station order by length(city),city limit 1;
```

Diamond java issue

```
public class Diamond extends BaseClass implements BaseInterface {

public static void main(String []args) {

new Diamond().foo();

}

}
```

No compiler error in this case: the compiler resolves to the definition in the class and the interface definition is ignored. This program prints “Base foo”. This can be considered as “class wins” rule

static variable initialized when class is loaded into JVM on the other hand instance variable has different value for each instances and they get created when instance of an object is created either by using new() operator or using reflection like Class.newInstance(). So if you try to access a non static variable without any instance compiler will complain because those variables are not yet created

Read more: http://javarevisited.blogspot.com/2012/02/why-non-static-variable-cannot-be.html#ixzz511ReVOXU

By using parametric queries and PreparedStatement you prevent many forms of SQL injection because all the parameters passed as part of place-holder will be escaped automatically by JDBC Driver

PreparedStatement allows you to write dynamic and parametric query.

JDBC best practice is that with auto commit mode disabled you can group SQL Statement in one transaction

ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property,

Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread

ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock

provides a method called lockInterruptibly()

ReentrantLock also provides convenient method to get List of all threads waiting for lock.

deadlock
where one thread holds lock on one object and waiting for other object lock which is held by other thread

Read more: http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html#ixzz511WfAW82

where one thread holds lock on one object and waiting for other object lock which is held by other thread

Race conditions occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other. Classical example of Race condition is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write

```
public Singleton getInstance(){
if(_instance == null){   //race condition if two threads sees _instance= null
_instance = new Singleton();
}
}

//race condition
if(!hashtable.contains(key)){
hashtable.put(key,value);
}
```

You can use Volatile variable if you want to read and write long and double variable atomically. long and double both are 64 bit data type and by default writing of long and double is not atomic and platform dependence

A volatile variable can be used as an alternative way of achieving synchronization in Java in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

volatile variable can be used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment

The Java volatile keyword cannot be used with method or class and it can only be used with a variable. Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable. By the way use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.

The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.

2. Synchronized obtains and releases the lock on monitor's Java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the volatile keyword in Java.

4. Synchronized method affects performance more than a volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.

6. You can not synchronize on the null object but your volatile variable in Java could be null.

7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

Read more: http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html#ixzz511cIpGuW

Kafka
three low-end machines can easily deal with two million writes per second
Data retention time in Kafka can be configured on a per-topic basis

* leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.

* "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
* "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

server.properties" and update the following this
1
2
3
4 $ vi ~/kafka/config/server.properties
broker.id=1 //Increase by one as per node count
host.name=x.x.x.x //Current node IP
zookeeper.connect=x.x.x.x:2181,x.x.x.x:2181,x.x.x.x:2181

There is no limit to the number of topics that can be used
– Topics can be created in advance, or created dynamically by Producers (see later)

Keys are used to determine which Partition
Each Partition is stored on the Broker's disk as one or more log files

Each message in the log is identified by its offset
– A monotonically increasing value
* The ability to add more Consumers to the system without reconfiguring the cluster
The Consumer Offset is stored in a special Kafka topic
– (Aside: Previously offsets were stored in ZooKeeper, but as of Kafka 9.0 this is no longer the case)

Kafka Brokers use ZooKeeper for a number of important internal features
– Leader election, failure detection

Kafka takes care of this by replicating each partition
– The replication factor is configurable
Kafka maintains replicas of each partition on other Brokers in the cluster
– Number of replicas is configurable
One Broker is the Leader for that Partition – All writes and reads go to and from the Leader – Other Brokers are Followers
▪ If a Leader fails, the Kafka cluster will elect a new Leader from among the

If there are multiple Partitions, you will not get total ordering when reading data
If you have a topic with three partitions, and ten Consumers in a Consumer Group reading that topic, only three Consumers will receive data
– One for each of the three Partitions

bootstrap.servers List of Broker host/port pairs used to establish the initial connection to the cluster
key.serializer Class used to serialize the key. Must implement the Serializer interface

value.serializer Class used to serialize the value. Must implement the Serializer interface
compression.type
acks=0: Producer will not wait for any acknowledgment from the server; acks=1: Producer will wait until the Leader has written the record to its local log; acks=all: Producer will wait until all in-sync replicas have acknowledged
ProducerRecord<String, String> record = new ProducerRecord<String, String>("mytopic", k, v);
1 4 producer.send(record); 5 6 producer.close();
group.id A unique string that identifies the Consumer Group this Consumer belongs to.
enable.auto.commit When set to true (the default)
ConsumerRecords<String, String> records = consumer.poll(100); 2 3 for
(ConsumerRecord<String, String> record : records) 4 System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.value());
* position(TopicPartition) provides the offset of the next record that will be fetched
If the number of Consumers changes, a partition rebalance occurs
– Partition ownership is moved around between the Consumers to spread the load evenly
– Consumers cannot consume messages during the rebalance, so this results in a short pause in message consumption
* commitSync() blocks until it succeeds What happens if you compare an object to null using equals()?
When a null object is passed as an argument to equals() method, it should return false, it must not throw NullPointerException, but if you call equals method on reference, which is null it will throw NullPointerException.

Read more: http://javarevisited.blogspot.com/2013/08/10-equals-and-hashcode-interview.html#ixzz516JZRNFq
key difference comes from the point that instanceof operator returns true, even if compared with subclass e.g. Subclass instanceof Superclass is true, but with getClass() it's false. By using getClass() you ensure that your equals() implementation doesn't return true if compared with subclass object

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，指的是函数的运行不依赖于外部变量或"状态"，只依赖于输入的参数
函数式编程不需要考虑"死锁"（deadlock），因为它不修改变量，所以根本不存在"锁"线程的问题。不必担心一个线程的数据，被另一个线程修改，所以可以很放心地把工作分摊到多个线程，部署"并发编程"

public static void topTwo(int[] numbers) { int max1 = Integer.MIN_VALUE; int max2 = Integer.MIN_VALUE; for (int number : numbers) { if (number > max1) { max2 = max1; max1 = number; } else if (number > max2) { max2 = number; } } System.out.println("Given integer array : " + Arrays.toString(numbers)); System.out.println("First maximum number is : " + max1); System.out.println("Second maximum number is : " + max2); }

Node jS

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。
当事件触发时，注册到这个事件的事件监听器被依次调用

```
var callback = function(stream) {
console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

setMaxListeners(n)
默认情况下， EventEmitters 如果你添加的监听器超过 10 个就会输出警告信息

java 8
Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8.
For example, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

java 8

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.
First solution is to create an own method that overrides the default implementation.

```
public class car implements vehicle, fourWheeler {

default void print() {
System.out.println("I am a four wheeler car vehicle!");
}
}
```

Second solution is to call the default method of the specified interface using super.

```
public class car implements vehicle, fourWheeler {

default void print() {
vehicle.super.print();
}
}
```

An interface can also have static helper methods from Java 8 onwards.
接口默认方法的继承分三种情况（分别对应上面的 InterfaceB 接口、InterfaceC 接口和 InterfaceD 接口）：
* 不覆写默认方法，直接从父接口中获取方法的默认实现。
* 覆写默认方法，这跟类与类之间的覆写规则相类似。
* 覆写默认方法并将它重新声明为抽象方法，这样新接口的子类必须再次覆写并实现这个抽象方法。

```
String phrase = persons
.stream()
.filter(p -> p.age >= 18)
.map(p -> p.name)
.collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

Lambda 表达式可以引用类成员和局部变量（会将这些变量隐式得转换成 final 的），例如下列两个代码块的效果完全相同：
函数式接口(Functional Interface)是 Java 8 对一类特殊类型的接口的称呼。 这类接口只定义了唯一的抽象方法的接口（除了隐含的 Object 对象的公共方法）， 因此最开始也就做 SAM 类型的接口（Single Abstract Method）。
JDK 中已有的一些接口本身就是函数式接口，如 Runnable。 JDK 8 中又增加了 java.util.function 包， 提供了常用的函数式接口。
java.util.function 中定义了几组类型的函数式接口以及针对基本数据类型的子接口。
* Predicate -- 传入一个参数，返回一个 bool 结果， 方法为 boolean test(T t)
* Consumer -- 传入一个参数，无返回值，纯消费。 方法为 void accept(T t)
* Function -- 传入一个参数，返回一个结果，方法为 R apply(T t)
* Supplier -- 无参数传入，返回一个结果，方法为 T get()
* UnaryOperator -- 一元操作符， 继承 Function,传入参数的类型和返回类型相同。
* BinaryOperator -- 二元操作符， 传入的两个参数的类型和返回类型相同， 继承 BiFunction

第二种方法引用的类型是静态方法引用，语法是 Class::static_method。注意：这个方法接受一个 Car 类型的参数。
```
cars.forEach( Car::collide );
```

```
// 获取当前的日期时间
LocalDateTime currentTime = LocalDateTime.now();
System.out.println("当前时间: " + currentTime);
```

```
// 22 小时 15 分钟
LocalTime date4 = LocalTime.of(22, 15);
System.out.println("date4: " + date4);
// 解析字符串
LocalTime date5 = LocalTime.parse("20:15:30");
System.out.println("date5: " + date5);


// 获取当前时间日期
ZonedDateTime date1 = ZonedDateTime.parse("2015-12-03T10:15:30+05:30[Asia/Shanghai]");
System.out.println("date1: " + date1);
```

kafka stream

state 使用的 local state, 效率更好， joined.aggregateBykey()-> state, 每个 statestore 存储互相独立的数据

Stream: alick:milk-> alice->eggs
Ktable: alice:milk changed to alice egge, updated result

alice 2->alice 3, Stream aggregate is 5, but stable aggregate is 3

一个新的参数 (MaxMetaspaceSize)可以使用。允许你来限制用于类元数据的本地内存。如果没有特别指定，元空间将会根据应用程序在运行时的需求动态设置大小

Stream Map java 8

```
// convert inside the map() method directly.
List<StaffPublic> result = staff.stream().map(temp -> {
StaffPublic obj = new StaffPublic();
obj.setName(temp.getName());
obj.setAge(temp.getAge());
if ("mkyong".equals(temp.getName())) {
obj.setExtra("this field is for mkyong only!");
}
return obj;
}).collect(Collectors.toList());
```

collector usage:

```
new Item("apple", 20, new BigDecimal("9.99"))
Map<String, Long> counting = items.stream().collect(
Collectors.groupingBy(Item::getName, Collectors.counting()));
```

filer null:
```
List<String> result = language.filter(Objects::nonNull).collect(Collectors.toList());

//Convert a Stream to List
List<String> result = language.collect(Collectors.toList());


//reuse a stream
Supplier<Stream<String>> streamSupplier = () -> Stream.of(array);
//get new stream
streamSupplier.get().forEach(x -> System.out.println(x));

//get another new stream
long count = streamSupplier.get().filter(x -> "b".equals(x)).count();
```

or else steam can not be rerun, got exception”
```
stream.forEach(x -> System.out.println(x));
// reuse it to filter again! throws IllegalStateException
long count = stream.filter(x -> "b".equals(x)).count();
```

List to Map:
To solve the duplicated key issue above, pass in the third mergeFunction argument like this :
```
Map<String, Long> result1 = list.stream().collect(
Collectors.toMap(Hosting::getName, Hosting::getWebsites,
(oldValue, newValue) -> oldValue
)
);
```

filter Map:
```
//Map -> Stream -> Filter -> MAP
Map<Integer, String> collect = A_MAP_EXAMPLE.entrySet().stream()
.filter(map -> map.getKey() == 2)
.collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));
```

flatMap:
```
Stream<String> lines = Arrays.stream(poetry.split("\n"));  Stream<String> words =
lines.flatMap(line -> Arrays.stream(line.split(" ")));
```

Optional.ofNullable() method returns a Non-empty Optional if a value present in the given object. Otherwise returns empty Optional.
Optional.isPresent() returns true if the given Optional object is non-empty.

```
StringJoiner sj = new StringJoiner("/", "prefix-", "-suffix");
sj.add("2016");
sj.add("02");
sj.add("26");
String result = sj.toString(); //prefix-2016/02/26-suffix
```

```
//java 8 read file into stream, try-with-resources
try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
stream.forEach(System.out::println);
} catch (IOException e) {
e.printStackTrace();
}
```

```
//convert
String password = "password123";
password.chars() //IntStream
.mapToObj(x -> (char) x)//Stream<Character>
.forEach(System.out::println);
```

Direct Approach (No Receivers)

区别于 Receiver-based 的数据消费方法，Spark 官方在 Spark 1.3 时引入了 Direct 方式的 Kafka 数据消费方式。相对于 Receiver-based 的方法，Direct 方式具有以下方面的优势：

简化并行(Simplified Parallelism)。不现需要创建以及 union 多输入源，Kafka topic 的 partition 与 RDD 的 partition 一一对应，官方描述如下：

由 Executor 读取数据，并参与到其他 Executor 的数据计算过程中去。driver 来决定读取多少 offsets，并将 offsets 交由 checkpoints 来维护。将触发下次 batch 任务，再由 Executor 读取 Kafka 数据并计算。从此过程我们可以发现 Direct 方式无需 Receiver 读取数据，而是需要计算时再读取数据

ECS, create docker repository first, then create the cluster.

CLoudformation , create template based on JSON (template, description, meta data, parameter, mapping, output, we can use cloudformation designer, to create template,
create stack,

Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. If each added method in an interface defined with implementation then no implementing class is affected. An implementing class can override the default implementation provided by the interface.

if is not in optional

Base 64.getHeader encoder

predicate is resuable filter, return boolean

```
return employees.stream().filter( predicate ).collect(Collectors.<Employee>toList());
public static Predicate<Employee> isAgeMoreThan(Integer age) {    return p -> p.getAge() > age;
  }
```

Ansible project notes:
1. scripts group_vars define variables
2. inventory define different server, when need to deploy to a new server, add one more file
3. roles define modules, like collector , media service
4. strategy , all host run task parallel
5. block, exception handling
6.

lsof -i :8080

netstat -tulpn

Scala grammer

def main(args: Array[String])

import java.awt._ // 引入包内所有成员

// 重命名成员
import java.util.{HashMap => JavaHashMap}

Any 是所有其他类的超类

var VariableName : DataType [= Initial Value]

在 Scala 中声明变量和常量不一定要指明数据类型，在没有指明数据类型的情况下，其数据类型是通过变量或常量的初始值推断出来的 var myVar = 10;

private[x]

如果写成 private[x],读作"这个成员除了对[…]中的类或[…]中的包中的类及它们的伴生对像可见外，对其它所有类都是 private

private[bobsrockets] class Navigator

2020/02/01
1.
== : 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()方法来两个对象的内容相等；若它们的内容相等，则返回 true(即，认为这两个对象相等)。

2.

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。另外需要注意的是： Object 的 hashcode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的内存地址 转换为整数之后返回

3. 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 fist 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

4.hashCode（）与 equals（）的相关规定

如果两个对象相等，则 hashcode 一定也是相同的
两个对象相等,对两个对象分别调用 equals 方法都返回 true
两个对象有相同的 hashcode 值，它们也不一定是相等的
因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖

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）

5.
String 类中使用 final 关键字修饰字符数组来保存字符串，private final char value[]，所以 String 对象是不可变的
StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
char[] value;
int count;
AbstractStringBuilder() { }
AbstractStringBuilder(int capacity)
{ value = new char[capacity]; }
```

StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁

性能
每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用

1. 操作少量的数据: 适用 String
2. 单线程操作字符串缓冲区下操作大量数据: 适用 StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据: 适用 StringBuffer

6.

String 真的是不可变的吗？

```
String str = "Hello";
      str = str + " World";
```
原来 String 的内容是不变的，只是 str 由原来指向"Hello"的内存地址转为指向"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串

7.

反射机制介绍

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一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

静态编译和动态编译

- **静态编译：**在编译时确定类型，绑定对象
- **动态编译：**运行时确定类型，绑定对象

反射机制优缺点

- **优点：** 运行期类型的判断，动态加载类，提高代码灵活度。
- **缺点：** 性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的 java 代码要慢很多。

我们在使用 JDBC 连接数据库时使用 Class.forName()通过反射加载数据库的驱动程序；② Spring 框架也用到很多反射机制，最经典的就是 xml 的配置模式。Spring 通过 XML 配置模式装载 Bean 的过程：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中;2)Java 类里面解析 xml 或 properties 里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反射机制，根据这个字符串获得某个类的 Class 实例; 4)动态配置实例的属性

```
  Class clazz = Class.forName("fanshe.Student");
Constructor[] conArray = clazz.getConstructors();
    //获取公有、无参的构造方法
Constructor con = clazz.getConstructor(null);

Object obj = con.newInstance();
```

8 JVM 运行机制

Java 程序从源代码到运行一般有下面 3 步：

我们需要格外注意的是 .class->机器码 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的(也就是所谓的热点代码)，所以后面引进了 JIT 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言.

Java 中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。**编译程序只需要面向虚拟机，生成虚拟机能够理解的代码**，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在 Java 中，这种供虚拟机理解的代码叫做字节码（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。**每一种平台的解释器（Interpreter）是不同的**，但是实现的虚拟机是相同的。Java 源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行

jvm convert .class to system specific code, that can be run in machine

9. JDK 和 JRE
JDK 是 Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。
JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。
如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet

10 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，抽象类可以有非抽象的方法
2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

注意：Java8 后接口可以有默认实现(default)。

11.

重载

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同.

重写

重写是子类对父类的允许访问的方法的实现过程进行重新编写,发生在子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类

Polymorphism
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
两种： 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）

12.

什么是线程和进程?

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。
在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程
线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的**堆**和**方法区**资源，但每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程

线程 是 进程 划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。线程执行开销小，但不利于资源的管理和保护 而进程正相反

13.
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 （静态变量，常量池,静态方法，类的代码）

- 直接内存

程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。**字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完。**

14 java 内存
https://github.com/Snailclimb/JavaGuide/blob/3965c02cc0f294b0bd3580df4868d5e396959e2e/Java%E7%9B%B8%E5%85%B3/%E5%8F%AF%E8%83%BD%E6%98%AF%E6%8A%8AJava%E5%86%85%E5%AD%98%E5%8C%BA%E5%9F%9F%E8%AE%B2%E7%9A%84%E6%9C%80%E6%B8%85%E6%A5%9A%E7%9A%84%E4%B8%80%E7%AF%87%E6%96%87%E7%AB%A0.md

15

什么是上下文切换?

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。
概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换**

16

认识线程死锁

下面通过一个例子来说明线程死锁,代码模拟了上图的死锁的情况 (代码来源于《并发编程之美》):
```
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();
}
}复制代码
```

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 复制代码
```

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

17 Threadlocal

在 ThreadLocalMap 的 set()，get() 和 remove() 方法中，都有清除无效 Entry 的操作，这样做是为了降低内存泄漏发生的可能。
https://juejin.im/post/5965ef1ff265da6c40737292#heading-7

在 set，get，initialValue 和 remove 方法中都会获取到当前线程，然后通过当前线程获取到 ThreadLocalMap，如果 ThreadLocalMap 为 null，则会创建一个 ThreadLocalMap，并给到当前线程。
在使用 ThreadLocal 类型变量进行相关操作时，都会通过当前线程获取到 ThreadLocalMap 来完成操作。**每个线程的 ThreadLocalMap 是属于线程自己的，ThreadLocalMap 中维护的值也是属于线程自己的。这就保证了 ThreadLocal 类型的变量在每个线程中是独立的，在多线程环境下不会相互影响。**

ThreadLocalMap 已经考虑到这种情况，并且有一些防护措施：在调用 ThreadLocal 的 get()，set() 和 remove() 的时候都会清除当前线程 ThreadLocalMap 中所有 key 为 null 的 value。这样可以降低内存泄漏发生的概率。所以我们在使用 ThreadLocal 的时候，每次用完 ThreadLocal 都调用 remove() 方法，清除数据，防止内存泄漏。

18
Why static methods cannot access non-static variables or methods?
Ans) A static method cannot access non-static variables or methods because static methods can be accessed without instantiating the class, so if the class is not instantiated the variables are not initialized and thus cannot be accessed from a static method.

19
java 8 https://www.java67.com/2018/10/java-8-stream-and-functional-programming-interview-questions-answers.html
What is the difference between Collection and Stream? (answer)
The main difference between a Collection and Stream is that Collection contains their elements but Stream doesn't. Stream work on a view where elements are actually stored by Collection or array, but unlike other views, any change made on Stream doesn't reflect on original collection.

20
parallelism
symbols.parallel()

.map(this::getStockInfo) //slow network operation

.collect(toList());
The problem is that all parallel streams use common fork-join thread pool, and if you submit a long-running task, you effectively block all threads in the pool

all tasks submitted to the common fork-join pool will not get stuck and finish in a reasonable time

21

Lamda 表达式作用域(Lambda Scopes)

访问局部变量

我们可以直接在 lambda 表达式中访问外部的局部变量：

```
@FunctionalInterface
public interface Converter<F, T> {
T convert(F from);
}

final int num = 1;
Converter<Integer, String> stringConverter =
(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3
```

但是和匿名对象不同的是，这里的变量 num 可以不用声明为 final，该代码同样正确：

```
int num = 1;
Converter<Integer, String> stringConverter =
(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3
```

不过这里的 num 必须不可被后面的代码修改（即隐性的具有 final 的语义），例如下面的就无法编译：

```
int num = 1;
Converter<Integer, String> stringConverter =
(from) -> String.valueOf(from + num);
num = 3;//在 lambda 表达式中试图修改 num 同样是不允许的。
```

与局部变量相比，我们对 lambda 表达式中的实例字段和静态变量都有读写访问权限。 该行为和匿名对象是一致的。

22
When your class implements java.io.Serializable interface it becomes Serializable in Java and gives compiler an indication that use Java Serialization mechanism to serialize this object

23
What is serialVersionUID? What would happen if you don't define this?

Consequence of not specifying serialVersionUID is that when you add or modify any field in class then already serialized class will not be able to recover because serialVersionUID generated for new class and for old serialized object will be different. Java serialization process

relies on correct serialVersionUID for recovering state of serialized object and
throws java.io.InvalidClassException in case of serialVersionUID mismatch

24
What will happen if one of the members in the class doesn't implement Serializable interface

but the object includes a reference to an non- Serializable class then a ‘NotSerializableException’ will be thrown at runtime

25
If a class is Serializable but its super class in not, what will be the state of the instance variables inherited from super class after deserialization?

Student extends Person, if persons doesn't implent serializable, then the field from 'person' will be null after deserialization.

26
Can you Customize Serialization process or can you override default Serialization process in Java?

We all know that for serializing an object ObjectOutputStream.writeObject (saveThisobject) is invoked and for reading object ObjectInputStream.readObject() is invoked but there is one more thing which Java Virtual Machine provides you is to define these two method in your class. If you define these two methods in your class then JVM will invoke these two methods instead of applying default serialization mechanism
private void writeObject(java.io.ObjectOutputStream stream)
throws IOException {
stream.writeObject(name);

}

27 Suppose super class of a new class implement Serializable interface, how can you avoid new class to being serialized?

If Super Class of a Class already implements Serializable interface in Java then its already Serializable in Java, since you can not unimplemented an interface its not really possible to make it Non Serializable class but yes there is a way to avoid serialization of new class. To avoid Java serialization you need to implement writeObject() and readObject() method in your Class and need to throw NotSerializableException from those method

28

Suppose you have a class which you serialized it and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?

It depends on whether class has its own serialVersionUID or not. As we know from above question that if we don't provide serialVersionUID in our code java compiler will generate it and normally it's equal to hashCode of object. by adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case Java Serialization API will throw java.io.InvalidClassException and this is the reason its recommended to have your own serialVersionUID in code and make sure to keep it same always for a single class.

29

Volatile variables have the visibility features of synchronized but not the atomicity features
The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory.
volatile modifier also provides visibility guarantee. Any change made to the volatile variable is visible to all threads. The value of the volatile variable is not cached by threads, instead, they are alway read from the main memory.

The volatile modifier also provides the **happens-before** guarantee, A write to volatile variable happens before any subsequent read. It also causes memory barrier to be flushed, which means all changes made by thread A before writing into the volatile variable will be visible to thread B when it read the value of the volatile field

it doesn't provide atomicity or mutual exclusion (synchronnized)

The key difference between volatile and synchronized modifier is a locking, the synchronized keyword need a lock but volatile is lock free

The second significant difference between synchronized and volatile modifier is *atomicity*, synchronized keyword provides the atomic guarantee and can make a block of code atomic but volatile variable doesn't provide such guarantee

30
抢占式

preemptive

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。
31
怎么检测一个线程是否持有对象监视器
Thread 类提供了一个 holdsLock(Object obj)方法，当且仅当对象 obj 的监视器被某条线程持有的时候才会返回 true，注意这是一个 static 方法，这意味着”某条线程”指的是当前线程。

future can check, task is done, task return result.
Future<String> future = executor.submit(callable);
//add Future to the list, we can get return value using Future
list.add(future);

CompletableFuture<String> receiver
= CompletableFuture.supplyAsync(this::findReceiver);

receiver.thenApplyAsync(this::sendMsg);
receiver.thenApplyAsync(this::sendMsg);
By using the *async suffix*, each message is submitted as separate tasks to the ForkJoinPool.commonPool(). This results in both the sendMsg callbacks being executed when the preceding calculation is done.

What to do when it all goes wrong

Luckily CompletableFuture has a nice way of handling this, using exceptionally.
CompletableFuture.supplyAsync(this::failingMsg)
.exceptionally(ex -> new Result(Status.FAILED))
.thenAccept(this::notify);

```
public void produce() throws InterruptedException
    {

        int value = 0;

        while (true) {

            synchronized (this)

            {

                // producer thread waits while list

                // is full

                while (list.size() == capacity)

                    wait();



                System.out.println("Producer produced-"

                                + value);
```

```
            // to insert the jobs in the list
            list.add(value++);

            // notifies the consumer thread that
            // now it can start consuming
            notify();

            // makes the working of program easier
            // to  understand
            Thread.sleep(1000);
        }
    }
}

// Function called by consumer thread
public void consume() throws InterruptedException
{
    while (true) {
        synchronized (this)
        {
            // consumer thread waits while list
            // is empty
```

```
            while (list.size() == 0)
                wait();

            // to retrive the ifrst job in the list
            int val = list.removeFirst();

            System.out.println("Consumer consumed-"
                               + val);

            // Wake up producer thread
            notify();

            // and sleep
            Thread.sleep(1000);
          }
        }
      }
```

33 什么导致线程阻塞
sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态
yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会
wait, nofify()这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法

的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 IllegalMonitorStateException

34

为什么 wait,nofity 和 nofityAll 这些方法不放在 Thread 类当中

一个很明显的原因是 JAVA 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的 wait()方法就有意义了。如果 wait()方法定义在 Thread 类中，线程正在等待的是哪个锁就不明显了。简单的说，由于 wait，notify 和 notifyAll 都是锁级别的操作，所以把他们定义在 Object 类中因为锁属于对象。

35

（1）ReentrantLock 可以对获取锁的等待时间进行设置，这样就避免了死锁 （2）ReentrantLock 可以获取各种锁的信息 （3）ReentrantLock 可以灵活地实现多路通知 另外，二者的锁机制其实也是不一样的:ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁，synchronized 操作的应该是对象头中 mark word.

36

如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务；如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue 满了，则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务

37

Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

38

什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功

39

CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

- CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某个点上之后，只是给某个数值-1 而已，该线程继续运行
- CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务
- CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就不可再用了

40 concurenthashmap

1. ConcurrentHashMap allows concurrent read and thread-safe update operation.

2. During the update operation, `ConcurrentHashMap` only locks a portion of Map instead of whole Map.

3. The concurrent update is achieved by internally dividing Map into the small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significantly higher number can be a waste of time and space and the lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of `ConcurrentHashMap` are thread-safe.

6. Since `ConcurrentHashMap` implementation doesn't lock whole Map, there is chance of read overlapping with update operations like `put()` and `remove()`. In that case result returned by `get()` method will reflect most recently completed operation from there start.

7. Iterator returned by `ConcurrentHashMap` is weekly consistent, fail-safe and never throw `ConcurrentModificationException`. In Java.

8. `ConcurrentHashMap` doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.

10. During `putAll()` and `clear()` operations, the concurrent read may only reflect insertion or deletion of some entries.

41

可以创建 Volatile 数组吗?

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了

42 join

```
Thread t1 = new Thread(new MyRunnable(), "t1");
Thread t2 = new Thread(new MyRunnable(), "t2");
Thread t3 = new Thread(new MyRunnable(), "t3");

t1.start();

//start second thread after waiting for 2 seconds or if it's dead
try {
t1.join(2000);
} catch (InterruptedException e) {
e.printStackTrace();
}

t2.start();

//start third thread only when first thread is dead
try {
t1.join();
} catch (InterruptedException e) {
e.printStackTrace();
}
```

43

Java 设计模式——实现单例模式的七种方式 [JZOF]

单例模式，框架中使用比较多的，开发中较为常用的设计模式。当某种业务需要保证只使用一种实例时，如 windows 中的任务管理器，永远只能打开一个窗口。

1、饿汉式，线程安全

```
public class Singleton{
//创建静态变量并进行初始化，并且 jvm 加载时就会进行实例，保证该实例只有一个
//但是对于多线程而言，这样的效率并不高，造成资源的浪费
private static final instance=new Singleton();
private Singleton(){}//将构造函数访问权限修改成 private，即外部无法通过 new 进行实例化调
用
public static Singleton getInstance(){
return instance;
}
}复制代码
```

2、懒汉式，线程不安全

```
//解决了饿汉式的资源浪费的缺点，但是只能在单线程下进行，如果多线程下会不安全
public class Singleton{
//创建静态变量，但不初始化，等待使用时进行初始化
private static Singleton instance=null;
private Singleton(){}

public static Singleton getInstance(){
if(instance==null){
instance=new Singleton();
}
return instance;
}
}复制代码
```

3、懒汉式，线程安全

```
//该线程安全版本，是在初始化方法上加上同步机制 synchronized，是的线程安全，但是这种全局同步，
会
使得在多线程并发情况下，执行效率不高，故而不推荐
public class Singleton{
private static Singleton instance=null;
private Singleton(){}

private synchronized Singleton getInstance(){
if(instance==null){
instance=new Singleton();
}
return instance;
}
}复制代码
```

4、懒汉式，变种，线程安全

```
//该线程安全版本，是在线程不安全版本上，使用静态块初始化 instance 实例，通过静态变量 jvm 初
始化规则
静态变量/静态块-非静态变量/构造块-构造函数
```

```
public class Singleton{
private static Singleton instance=null;
static {
instance=new Singleton();
}
private Singleton(){}
public static Singleton getInstance(){
return instance;
}
}复制代码
```

5、使用静态内部类实现，线程安全【推荐】

```
public class Singleton{
private static final class SingletonHolder{
private static final Singleton instance=new Singleton();
}
private Singleton(){}
public static Singleton getInstance(){
return SingletonHolder.instance;
}
}复制代码
```

6、使用枚举，线程安全【推荐】

```
public class Singleton{
private Singleton(){}
private static Singleton getInstance(){
return SingletonHolder.INSTANCE.getInstance;
}
public enum SingletonHolder{
INSTANCE;
private Singleton instance=null;
SingletonHolder(){
instance=new Singleton();
}
public Singleton getInstance(){
return instance;
}
}
}复制代码
```

7、双重校验锁，线程安全【推荐】

```
public class Singleton{
private static Singleton instance=null;
private Singleton(){}
public static Singleton getInstance(){
synchronized(Singleton.class){
if(instance==null){
instance=new Singleton();
}
return instance;
```

```
}
}
```

有两个问题需要注意：

1、如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些 servlet 容器对每个 servlet 使用完全不同的类 装载器，这样的话如果有两个 servlet 访问一个单例类，它们就都会有各自的实例。

2、如果 Singleton 实现了 java.io.Serializable 接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是：

```
1 private static Class getClass(String classname)
2                                 throws ClassNotFoundException {
3     ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
4
5     if(classLoader == null)
6       classLoader = Singleton.class.getClassLoader();
7
8     return (classLoader.loadClass(classname));
9   }
10 }
11
```

对第二个问题修复的办法是：

```
1 public class Singleton implements java.io.Serializable {
2   public static Singleton INSTANCE = new Singleton();
3
4   protected Singleton() {
5
6   }
7   private Object readResolve() {
8         return INSTANCE;
9     }
10 }
```

44

Collection interface defines `remove(Object obj)` method to remove objects from Collection. List interface adds another method `remove(int index)`, which is used to remove object at specific index. You can use any of these method to remove an entry from Collection, while not iterating. Things change, when you iterate. Suppose you are traversing a List and removing only certain elements based on logic, then you need to use Iterator's `remove()` method. This method removes current element from Iterator's perspective. If you use Collection's or List's `remove()` method during iteration then your code

will throw `ConcurrentModificationException`. That's why it's advised to use Iterator remove() method to remove objects from Collection

45

Iterator duplicate functionality of `Enumeration` with one addition of `remove()` method and both provide navigation functionally on objects of Collection.Another difference is that Iterator is more safe than `Enumeration` and doesn't allow another thread to modify collection object during iteration except remove() method and throws `ConcurrentModificaitonException`. See Iterator vs

46

If you look at source code of `java.util.HashSet` class, you will find that that it uses a HashMap with same values for all keys, as shown below:

```
private transient HashMap map;

// Dummy value to associate with an Object in the backing Map
private static final Object PRESENT = new Object();
```

When you call `add()` method of `HashSet`, it put entry in HashMap :

```
public boolean add(E e) {
  return map.put(e, PRESENT)==null;
}
```

HashSet 和 HashMap 一样也需要实现 hash 算法来计算对象的 hash 值，但不同的是，HashMap 中添加一个键值对的时候， (Key, Value)，hash 函数计算的是 Key 的 hash 值。而

HashSet 则是计算 value 的 hash 值。当我们调用 HashSet 的 add（E e）的方法 的时候，我们会计算机元素 e 的 hash 值

47

The Hidden Contract Between equals and Comparable

HashMap, ArrayList, and **HashSet** add elements based on the **equals** method,

TreeMap and **TreeSet** are ordered and use the **compareTo** method

If two objects are considered equal by the* equals *operation, then try to make those objects must equal by the* Comparable *or* Comparator *test.

48

hashMap ,null key value allowd,

HashMap is a fail-fast iterator while the enumerator for the Hashtable is not

49

1) `List` maintains insertion order of elements while `Set` doesn't maintain any ordering.

2) `The list` allows duplicate objects while `Set` doesn't allow any duplicates.

3) list position access, set not,

50

How do you Sort objects on the collection?

Sorting is implemented using Comparable and Comparator in Java and when you call Collections.sort() it gets sorted based on the natural order specified in `compareTo()` method while Collections.sort(Comparator) will sort objects based on compare() method of Comparator

51.

First and most common *difference between Vector vs ArrayList* is that Vector is synchronized and thread-safe while `ArrayList` is neither Synchronized nor thread-safe

`Vector` vs `ArrayList` is Speed, which is directly related to previous difference. Since Vector is synchronized, its slow and ArrayList is not synchronized its faster than Vector.

Third difference on Vector vs ArrayList is that Vector is a legacy class and initially it was not part of Java Collection Framework

52.

Though ArrayList is also backed up by array, it offers some usability advantage over array in Java. Array is fixed length data structure, once created you can not change it's length. On the other hand, ArrayList is dynamic, it automatically allocate a new array and copies content of old array, when it resize. Another reason of using ArrayList over Array is support of Generics. Array doesn't support Generics

53

Can we replace Hashtable with ConcurrentHashMap? (answer)
Answer 3: Yes we can replace Hashtable with ConcurrentHashMap and that's what suggested in Java documentation of ConcurrentHashMap. but you need to be careful with code which relies on locking behavior of Hashtable. Since Hashtable locks whole Map instead of a portion of Map, compound operations like if(Hashtable.get(key) == null) put(key, value) works in Hashtable but not in concurrentHashMap. instead of this use putIfAbsent() method of ConcurrentHashMap

putifAbsent() return null, if the value doesn't exist before.

54

copyonwrite list

CopyOnWriteArrayList 就是通过 Copy-On-Write(COW)，即写时复制的思想来通过延时更新的策略来实现数据的最终一致性，并且能够保证读线程间不阻塞。
COW 通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。对 CopyOnWrite 容器进行并发的读的时候，不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，延时更新的策略是通过在写的时候针对的是不同的数据容器来实现的，放弃数据实时性达到数据的最终一致性。

ist implementation ArrayList, LinkedList and CopyOnWriteArrayList in Java:

To solve this problem you can use Java's volatile keyword. The `volatile` keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated

64 **Synchronized**
You can use java synchronized keyword only on synchronized method or synchronized block. thread enters into java synchronized method or blocks it **acquires a lock** and whenever it leaves java synchronized method or block it releases the lock.

The lock is released even if thread leaves synchronized method after completion or due to any Error or Exception

Java Thread acquires an **object level lock** when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method

Java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires the same lock then the current thread which is holding lock can enter into that method without acquiring the lock.

disadvantage of Java synchronized keyword is that it doesn't allow concurrent read

Java synchronized block is better than java synchronized method in Java because by using synchronized block you can only lock critical section of code and avoid locking the whole method which can possibly degrade performance

Here is how double checked locking looks like in Java :

```
public static Singleton getInstanceDC() {
if (_instance == null) { // Single Checked
synchronized (Singleton.class) {
if (_instance == null) { // Double checked
_instance = new Singleton();
}
}
}
return _instance;
}
```

On surface this method looks perfect, as you only need to pay price for synchronized block one time, but it still broken, until you make _instance variable volatile.

Without volatile modifier it's possible for another thread in Java to see half initialized state of _instance variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile _instance before any read of _instance variable.

Java synchronized block is better than java synchronized method in Java because by using synchronized block you can only lock critical section of code and avoid locking the whole method which can possibly degrade performance

It's possible that both static synchronized and non-static synchronized method can run simultaneously or concurrently because they lock on the different object.

According to the Java language specification **you can not use Java synchronized keyword with constructor** it's illegal and result in compilation error.

Do not synchronize on the non-final field on synchronized block in Java. because the reference of the non-final field may change anytime and then different thread might synchronizing on different objects

It's **not recommended to use String object as a lock in java synchronized block** because a string is an immutable object and literal string and interned string gets stored in String pool. so by any chance if any other part of the code or any third party library used same String as there lock then they both will be locked on the same object despite being completely unrelated which could result in unexpected behavior and bad performance. instead of String object its advised to use new Object() for **Synchronization in Java on synchronized block**.

```
private static final String LOCK = "lock"; //not recommended
private static final Object OBJ_LOCK = new Object(); //better

public void process() {
synchronized(LOCK) {
.....
}
}
```

65 Join()

:It will put the current thread on wait until the thread on which it is called is dead or wait for specified time

```
// creating two threads
        ThreadJoining t1 = new ThreadJoining();
        ThreadJoining t2 = new ThreadJoining();
        ThreadJoining t3 = new ThreadJoining();

        // thread t1 starts
        t1.start();

        // starts second thread after when
        // first thread t1 has died.
        try
        {
            System.out.println("Current Thread: "
                  + Thread.currentThread().getName());
            t1.join();
        }

        catch(Exception ex)
        {
            System.out.println("Exception has " +
                               "been caught" + ex);
        }

        // t2 starts
        t2.start();
```

66

The advantages of a lock are

- it’s possible to make them fair
- it’s possible to make a thread responsive to interruption while waiting on a Lock object.

- it’s possible to try to acquire the lock, but return immediately or after a timeout if the lock can’t be acquired
- it’s possible to acquire and release locks in different scopes, and in different orders

The major advantage of lock interfaces on multi-threaded and concurrent programming is they provide two separate lock for reading and writing which enables you to write high-performance data structure like ConcurrentHashMap

67
The only major difference is that wait releases the lock or monitor while sleep doesn't release any lock or monitor while waiting. The wait is used for inter-thread communication while sleep is used to introduce pause on execution

68

An atomic operation is an operation which is performed as a single unit of work without the possibility of interference from other operations.

The Java language specification guarantees that reading or writing a variable is an atomic operation(unless the variable is of type `long` or `double`). Operations variables of type `long` or `double` are only atomic if they declared with the `volatile` keyword.

69
race condition

```
public Singleton getInstance(){

if(_instance == null){   //race condition if two threads sees _instance= null

_instance = new Singleton();

}

}
```

read-modify-update race conditions

```
if(!hashtable.contains(key)){

hashtable.put(key,value);

}
```

70

How will you awake a blocked thread in Java?
This is a tricky question on threading, blocking can result in many ways, if thread is blocked on IO then I don't think there is a way to interrupt the thread, let me know if there is any, on the other hand, if thread is blocked due to result of calling wait(), sleep(), or join() method you can interrupt the thread and it will awake by throwing InterruptedException

`Thread.sleep` throws `InterruptedException` if the thread is interrupted during the sleep. Catch that, and check your flag.

71

CountDownLatch

countDown()

Decrements the count of the latch, releasing all waiting threads if
* the count reaches zero.

```
@Override
    public void run()
    {
        try
        {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(Thread.currentThread().getName()
                               + " finished");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
```

```
// Let us create task that is going to
        // wait for four threads before it starts
        CountDownLatch latch = new CountDownLatch(4);

        // Let us create four worker
        // threads and start them.
        Worker first = new Worker(1000, latch,
                                  "WORKER-1");
        Worker second = new Worker(2000, latch,
                                  "WORKER-2");
        Worker third = new Worker(3000, latch,
                                  "WORKER-3");
        Worker fourth = new Worker(4000, latch,
                                  "WORKER-4");
        first.start();
        second.start();
        third.start();
        fourth.start();

        // The main task waits for four threads
        latch.await();
```

72

Comparable and Comparator in Java and when you call Collections.sort() it gets sorted based on the natural order specified in compareTo() method while Collections.sort(Comparator) will sort objects based on compare() method of Comparator.

73

Lock lock = new ReentrantLock();

进入 get 后，如果获得了锁，调用 set 的时候不需要再获得锁，可以直接用

get(){

lock.lock()

set();

}

set(){

```
lock.lock()

}
```

74 why variable in lambda is final

Local Variables in Capturing Lambdas
Simply put, this won't compile:

```
Supplier<Integer> incrementer(int start) {
return () -> start++;
}
```

start is a local variable, and we are trying to modify it inside of a lambda expression.

The basic reason this won't compile is that the lambda is capturing the value of start, meaning making a copy of it. Forcing the variable to be final avoids giving the impression that incrementing start inside the lambda could actually modify the start method parameter.

But, why does it make a copy? Well, notice that we are returning the lambda from our method. Thus, the lambda won't get run until after the start method parameter gets garbage collected. Java has to make a copy of start in order for this lambda to live outside of this method.

3.1. Concurrency Issues
For fun, let's imagine for a moment that Java did allow local variables to somehow remain connected to their captured values.

What should we do here:

```
public void localVariableMultithreading() {
```

```
boolean run = true;
executor.execute(() -> {
while (run) {
// do operation
}
});

run = false;
}
```

While this looks innocent, it has the insidious problem of “visibility”. Recall that each thread gets its own stack, and so how do we ensure that our while loop sees the change to the run variable in the other stack? The answer in other contexts could be using synchronized blocks or the volatile keyword.

However, because Java imposes the effectively final restriction, we don't have to worry about complexities like this.

75

总的来说，Lock 和 Synchronized 有以下几点不同：

(1). Lock 是一个接口，是 JDK 层面的实现；而 synchronized 是 Java 中的关键字，是 Java 的内置特性，是 JVM 层面的实现；

(2). synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象，因此使用 Lock 时需要在 finally 块中释放锁；

(3). Lock 可以让等待锁的线程响应中断，而使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断；

(4). 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到；

(5). Lock 可以提高多个线程进行读操作的效率。

76

若内存地址不变, 则值也不可以改变的东西称为常量，典型的 String 就是不可变的，所以称之为 常量(constant)。此外，我们可以通过 final 关键字来定义常量，但严格来说，只有基本类型被其修饰后才是常量（对基本类型来说是其值不可变，而对于对象变量来说其引用不可再变）。

77 String

不同字符串可能共享同一个底层 char 数组，例如字符串 String s=”abc” 与 s.substring(1) 就共享同一个 char 数组：char[] c = {‘a’,’b’,’c’}。其中，前者的 offset 和 count 的值分别为 0 和 3，后者的 offset 和 count 的值分别为 1 和 2。

78
不能改变状态指的是不能改变对象内的成员变量，包括：

基本数据类型的值不能改变;

引用类型的变量不能指向其他的对象;

引用类型指向的对象的状态也不能改变;

除此之外，还应具有以下特点：

除了构造函数之外，不应该有其它任何函数（至少是任何 public 函数）修改任何成员变量;

任何使成员变量获得新值的函数都应该将新的值保存在新的对象中，而保持原来的对象不被修改。

79
，value，offset 和 count 这三个变量都是 final 的， 也就是说在 String 类内部，一旦这三个值初始化了， 也不能被改变。所以，可以认为 String 对象是不可变的了。

那么在 String 中，明明存在一些方法，调用他们可以得到改变后的值。这些方法包括 substring， replace， replaceAll， toLowerCase 生产 new string

80 反射改变 String

```
public final class String
implements java.io.Serializable, Comparable<string>, CharSequence
{
/** The value is used for character storage. */
private final char value[];

/** The offset is the first index of the storage that is used. */
private final int offset;

/** The count is the number of characters in the String. */
private final int count;

String s = "Hello World";

System.out.println("s = " + s); //Hello World
```

```
//获取 String 类中的 value 字段
Field valueFieldOfString = String.class.getDeclaredField("value");

//改变 value 属性的访问权限
valueFieldOfString.setAccessible(true);

//获取 s 对象上的 value 属性的值
char[] value = (char[]) valueFieldOfString.get(s);

//改变 value 所引用的数组中的第 5 个字符
value[5] = '_';

System.out.println("s = " + s); //Hello_World
```

81.

对于 String str=new String(“abc”)：如果常量池中原来没有”abc”，则会产生两个对象（一个在常量池中，一个在堆中）；否则，产生一个对象。

82

在泛型类中， static 域或方法无法访问泛型类的类型参数(类型实例化所得的所有泛型类型共享同一个 Class 对象)；若静态方法需要使用泛型能力，就必须使其成为泛型方法（不与泛型类共享类型参数）。

在一个类中，static 域或方法都是该类的 Class 对象的成员，而我们知道泛型所创造出来的所有类型都共享一个 Calss 对象， 因此实质上不受泛型参数限制，所以如下代码根本不能通过编译:

```
public class Test2<T> {
public static T one; //编译错误
public static T show(T one){ //编译错误
return null;
}
}
1
2
3
4
5
6
```

但是要注意区分下面的一种情况：

```
public class Test2<T> {
public static <T> T show(T one){//这是正确的
return null;
}
}
```

82

static 域或方法无法访问泛型类的类型参数(类型实例化所得的所有泛型类型共享同一个 Class 对象)；若静态方法需要使用泛型能力，就必须使其成为泛型方法

```
public class Test2<T> { public static <T> T show(T one){//这是正确的  return null; } }
```

默认情形下任何类型都可以作为参数插入。特别地，为类型参数设定的第一个边界可以是类类型或接口类型,类型参数的第一个边界之后的任意额外边界都只能是接口类型，同时，一般将标记性接口放到靠后位置,这些类型参数之间有 & 相连接。

```
publc class MyClass<T extends Number & Serilizable>{
...
}
```

83

在 Java 集合框架中，对于参数值是未知类型的容器类，只能读取其中元素，不能向其中添加元素

```
ArrayList<?> list = new ArrayList<String>();
list = new ArrayList<Double>();

list.add(e); // e cannot be resolved to a variable
System.out.println(list1.size()); // OK
```

- List<Object> ： 编译器认为 List<Object> 是 List<?> 的子类型；

泛型的主要目标之一就是将这种错误检查移到编译期

84.

flist 的类型就是 List<? extends Fruit>了，但这并不意味着可以向这个 List 可以添加任何类型的 Fruit，甚至于不能添加 Apple。虽然编译器知道这个 List 持有的是 Fruit，但并不知道其具体持有哪种特定类型(可能是 List<Fruit>，List<Apple>，List<Orange>，List<Jonathan>)，所以编译器不知道该添加那种类型的对象才能保证类型安全

```
public class GenericTest {
class Fruit {}
class Apple extends Fruit {}
public void test(){
Fruit fruit = new Fruit();
Apple apple = new Apple();
List<? extends Fruit> flist = new ArrayList<>();
//error below compile error
//flist.add(apple);

//ok here
      Fruit f = flist.get(0);
flist.contains(new Apple()); // OK
flist.indexOf(new Apple()); // OK

}
}


List<? super Apple> appleList = new ArrayList<>();
appleList.add(apple);
//error, 不允许向该 List 放入一个 Fruit 对象， 因为该 List 的类型可能是 List<Apple> ,
// 这样将会违背泛型的本意
appleList.add(fruit);
```

PECS: producer(读取)-extends, consumer(写入)-super。换句话说，如果输入参数表示一个 T 的生产者，就使用<? extends T>；如果输入参数表示一个 T 的消费者，就使用<? super T>

85

CopyOnWriteArrayList 和 Collections.synchronizedList 是实现线程安全的列表的两种方式。两种实现方式分别针对不同情况有不同的性能表现，其中 CopyOnWriteArrayList 的写操作性能较差(复制数组)，而多线程的读操作性能较好；而 Collections.synchronizedList 的写操作性能比 CopyOnWriteArrayList 在多线程操作的情况下要好很多，而读操作因为是采用了 synchronized 关键字的方式，其读操作性能并不如 CopyOnWriteArrayList。因此在不同的应用场景下，应该选择不同的多线程安全实现类。

https://blog.csdn.net/qq_20492405/article/details/103595186

86

为什么使用 CAS 代替 synchronized

synchronized 加锁，同一时间段只允许一个线程访问，能够保证一致性但是并发性下降。而是用 CAS 算法使用 do-while 不断

判断而没有加锁（实际是一个自旋锁），保证一致性和并发性。

原子性保证：CAS 算法依赖于 rt.jar 包下的 sun.misc.Unsafe 类，该类中的所有方法都是 native 修饰的，直接调用操作
系统底层资源执行相应的任务。

ABA 问题，一个线程改动后，再改回去，另外的一个县一个线程不知道。

87

```
Callable callable = new CallableExample();
        // Create the FutureTask with Callable
      randomNumberTasks[i] = new FutureTask(callable);

      // As it implements Runnable, create Thread
      // with FutureTask
      Thread t = new Thread(randomNumberTasks[i]);
      t.start();
```

randomNumberTasks[i].get()

88

```
/**
* runAsync 方法也好理解，它以 Runnable 函数式接口类型为参数，所以
CompletableFuture 的计算结果为空。

supplyAsync 方法以 Supplier 函数式接口类型为参数,CompletableFuture 的计算结果类型为
U。
* @param args
 * @throws Exception
*/
public static void main(String[] args) throws Exception {
Runnable runnable1 = new Runnable() {
@Override
      public void run() {
System.out.println("Thread name : " + Thread.currentThread().getName());
}
};

CompletableFuture future1 = CompletableFuture.runAsync(runnable1);
future1.whenComplete((v, e) -> {
System.out.println("v1: " + v);
System.out.println("e1: " + e);
});

CompletableFuture<Integer> future =
CompletableFuture.supplyAsync(CompletableFutureTest::getMoreData);
//bicomsumer
// BiConsumer is a functional interface; it takes two arguments and returns nothing.
```

```
   Future<Integer> f = future.whenComplete((v, e) -> {
System.out.println("v: " + v);
System.out.println("e: " + e);
});
System.out.println(f.get());
}
```

89Rutime.getAvailableProcessors();

90

而使用了 volatile 关键字之后，情况就有所不同，volatile 关键字有两层语义：

1、立即将缓存中数据写会到内存中

2、其他处理器通过嗅探总线上传播过来了数据监测自己缓存的值是不是过期了，如果过期了，就会对应的缓存中的数据置为无效。而当处理器对这个数据进行修改时，会重新从内存中把数据读取到缓存中进行处理。

在这种情况下，不同的 CPU 之间就可以感知其他 CPU 对变量的修改，并重新从内存中加载更新后的值，因此可以解决可见性问题。

91
所以 value 也只是一个引用，它指向一个真正的数组对象。其实执行了 String s = “ABCabc”; 这句代码之后，真正的内存布局应该是这样的：

value，offset 和 count 这三个变量都是 private 的，并且没有提供 setValue， setOffset 和 setCount 等公共方法来修改这些值，所以在 String 类的外部无法修改 String。也就是说一旦初始化就不能修改， 并且在 String 类的外部不能访问这三个成员。此外，value，offset 和 count 这三个变量都是 final 的， 也就是说在 String 类内部，一旦这三个值初始化了，也不能被改变。所以可以认为 String 对象是不可变的了

92

ArrayList
1）Object[] elementData：数据存储
2）int size：使用数量
3）int modCount：操作次数
4）初始化：
a、指定容量初始化数组；
b、不指定容量第一次 add 数据时初始化数组容量 10

5）扩容：
a、1.5 倍；
b、不够取所需最小；
c、新容量大于 MAX_ARRAY_SIZE（Integer.MAX_VALUE-8）,按所需容量取 MAX_ARRAY_SIZE 和 Integer.MAX_VALUE 较小值

linkedList
1) Node {E item, Node prev, Node next}
2) int size
3) Node first
4) Node last
5) linkFirst(), linkLast(), linkBefore(), unLinkFirst(), unLinkLast(), unLink(), indexOf()
6）int modCount

Object
1) wait(), notify(), notifyAll(), wait(timeout) 2) hashCode(), equals() 3) clone()

1. broker 配置

server.property, 可以配置 id， port, 如果一个机器配置多个 broker, 端口一定要不一样

replication factor, 决定 了， 消息在哪个节点上复制， topic desc 可以查看

1.0 里面，消息读写都是从 Leader。

生产环境

file handler

log rolling

advertised.listeners :external IP on which rest of the work can connect to your cluster

monitor lag

0.8.x 版中，Consumer 使用 Apache ZooKeeper 来协调 Consumer group，而许多已知的 Bug 会导致其长期处于再均衡状态

VM 上运行各种 Consumers 时，请警惕垃圾回收对它们可能产生的影响, 长时间垃圾回收的停滞，可能导致 ZooKeeper 的会话被丢弃、或 Consumer group 处于再均衡状态

为各个 Producer 配置 Retries ,其默认值为 3，当然是非常低的

单个 Leader 所使用的网络 I/O，至少是 Follower 的四倍。而且，Leader 还可能需要对磁盘进行读操作，而 Follower 只需进行写操作

按需修改 Apache Log4j 的各种属性

对于具有高吞吐量服务级别目标（service level objectives，SLOs）的大型群集，请考虑为 Brokers 的子集隔离出不同的 Topic

.index offset

.log file

resetoffset by time

```
kafka-consumer-groups --bootstrap-server <kafkahost:port> --group <group_id> --topic
<topic_name> --reset-offsets --to-earliest --execute
```

- --shift-by <positive_or_negative_integer>
- --to-current
- --to-latest
- --to-offset <offset_integer>
- --to-datetime <datetime_string>
- --by-duration <duration_string>

- The primary advantage of using EBS in a Kafka deployment is that it significantly reduces data-transfer traffic when a broker fails or must be replaced. The replacement broker joins the cluster much faster.
- Data stored on EBS is persisted in case of an instance failure or termination. The broker’s data stored on an EBS volume remains intact, and you can mount the EBS volume to a new EC2 instance

Schema:

handle schema /add/remove filed

AVro serialiable/des,

Avro: 1) define schema 2) serialble the code based on the schema 3) extract the schema info and deserialible the information

avro toll can generate java class based on schema

KafkaAvroSerilizer store the schemaid to Schema registry service, KafkaAvroDeserializer get the id , and deserialize the data

Schema compatibility checking is implemented in Schema Registry by versioning every single schema. The compatibility type determines how Schema Registry compares the new schema with previous versions of a schema, for a given subject. When a schema is first created for a subject, it gets a unique id and it gets a version number

commited offset: consumer already processed. after rebalancing, the consumer start from 'committed offset'

auto/manu commit

'auto.commit.interval.ms': default 5 second. commit 的间隔时间， 即使 consumer 处理完了，时间没到，也没 commit

manu commit: commit Sync/commit Async (no retry)

rebalance: process too slow, or other reason, in this case, consumer want to commit what have been done.

maintain an offset of current ,

consumerRebalancerListener()-{

onPartitionrevoed （before reparition happed）repartion 前可以 commit offset

onParutitionAssigned(afater get a new partition

}

ReblanacerListener reblancer = new ReblanceListner(consumer);

8.

consumer consume msg from some defined partition (like first 3 partitions)

cosnumer.assign(partitionList), then consumer will consume msg from these partition list.

注意：

https://juejin.im/entry/5bf4de8ce51d4560336cc28a

配置

kafka 配置文件详细

转载独家记忆 shine 最后发布于 2017-11-29 11:40:27 阅读数 2862 收藏

展开

1：producer.properties

```
#指定 kafka 节点列表，用于获取 metadata，不必全部指定
```

#需要 kafka 的服务器地址，来获取每一个 topic 的分片数等元数据信息。
metadata.broker.list=kafka01:9092,kafka02:9092,kafka03:9092#生产者生产的消息被发送到哪个 block，需要一个分组策略。

#指定分区处理类。默认 kafka.producer.DefaultPartitioner，表通过 key 哈希到对应分区

#partitioner.class=kafka.producer.DefaultPartitioner#生产者生产的消息可以通过一定的压缩策略（或者说压缩算法）来压缩。消息被压缩后发送到 broker 集群，

#而 broker 集群是不会进行解压缩的，broker 集群只会把消息发送到消费者集群，然后由消费者来解压缩。

#是否压缩，默认 0 表示不压缩，1 表示用 gzip 压缩，2 表示用 snappy 压缩。

#压缩后消息中会有头来指明消息压缩类型，故在消费者端消息解压是透明的无需指定。

#文本数据会以 1 比 10 或者更高的压缩比进行压缩。compression.codec=none

#指定序列化处理类，消息在网络上传输就需要序列化，它有 String、数组等许多种实现。
serializer.class=kafka.serializer.DefaultEncoder

#如果要压缩消息，这里指定哪些 topic 要压缩消息，默认 empty，表示不压缩。

#如果上面启用了压缩，那么这里就需要设置

#compressed.topics=

#这是消息的确认机制，默认值是 0。在面试中常被问到。

#producer 有个 ack 参数，有三个值，分别代表：

#（1）不在乎是否写入成功；

#（2）写入 leader 成功；

#（3）写入 leader 和所有副本都成功；

#要求非常可靠的话可以牺牲性能设置成最后一种。#为了保证消息不丢失，至少要设置为 1，也就

#是说至少保证 leader 将消息保存成功。

#设置发送数据是否需要服务端的反馈,有三个值 0,1,-1，分别代表 3 种状态：

#0: producer 不会等待 broker 发送 ack。生产者只要把消息发送给 broker 之后，就认为发送成功了，这是第 1 种情况；

#1: 当 leader 接收到消息之后发送 ack。生产者把消息发送到 broker 之后，并且消息被写入到本地文件，才认为发送成功，这是第二种情况；#-1: 当所有的 follower 都同步消息成功后发送 ack。不仅是主的分区将消息保存成功了，

#而且其所有的分区的副本数也都同步好了，才会被认为发动成功，这是第 3 种情况。
request.required.acks=0#broker 必须在该时间范围之内给出反馈，否则失败。

#在向 producer 发送 ack 之前,broker 允许等待的最大时间 ，如果超时,

#broker 将会向 producer 发送一个 error ACK.意味着上一次消息因为某种原因

#未能成功(比如 follower 未能同步成功)request.timeout.ms=10000#生产者将消息发送到 broker，有两种方式，一种是同步，表示生产者发送一条，broker 就接收一条；

#还有一种是异步，表示生产者积累到一批的消息，装到一个池子里面缓存起来，再发送给 broker，

#这个池子不会无限缓存消息，在下面，它分别有一个时间限制（时间阈值）和一个数量限制（数量阈值）的参数供我们来设置。

#一般我们会选择异步。

#同步还是异步发送消息，默认“sync”表同步，"async"表异步。异步可以提高发送吞吐量,

#也意味着消息将会在本地 buffer 中,并适时批量发送，但是也可能导致丢失未发送过去的消息
producer.type=sync

#在 async 模式下,当 message 被缓存的时间超过此值后,将会批量发送给 broker,

#默认为 5000ms

#此值和 batch.num.messages 协同工作.queue.buffering.max.ms = 5000#异步情况下，缓存中允许存放消息数量的大小。

#在 async 模式下,producer 端允许 buffer 的最大消息量

#无论如何,producer 都无法尽快的将消息发送给 broker,从而导致消息在 producer 端大量沉积

#此时,如果消息的条数达到阀值,将会导致 producer 端阻塞或者消息被抛弃，默认为 10000 条消息。
queue.buffering.max.messages=20000#如果是异步，指定每次批量发送数据量，默认为
200batch.num.messages=500#在生产端的缓冲池中，消息发送出去之后，在没有收到确认之前，该缓冲池中的消息是不能被删除的，

#但是生产者一直在生产消息，这个时候缓冲池可能会被撑爆，所以这就需要有一个处理的策略。

#有两种处理方式，一种是让生产者先别生产那么快，阻塞一下，等会再生产；另一种是将缓冲池中的消息清空。

#当消息在 producer 端沉积的条数达到"queue.buffering.max.meesages"后阻塞一定时间后,

#队列仍然没有 enqueue(producer 仍然没有发送出任何消息)

#此时 producer 可以继续阻塞或者将消息抛弃,此 timeout 值用于控制"阻塞"的时间

#-1: 不限制阻塞超时时间，让 produce 一直阻塞,这个时候消息就不会被抛弃

#0: 立即清空队列,消息被抛弃 queue.enqueue.timeout.ms=-1#当 producer 接收到 error ACK,或者没有接收到 ACK 时,允许消息重发的次数

#因为 broker 并没有完整的机制来避免消息重复,所以当网络异常时(比如 ACK 丢失)

```
#有可能导致 broker 接收到重复的消息,默认值为 3.message.send.max.retries=3#producer 刷新 topic metada
的时间间隔,producer 需要知道 partition leader

#的位置,以及当前 topic 的情况

#因此 producer 需要一个机制来获取最新的 metadata,当 producer 遇到特定错误时,

#将会立即刷新

#(比如 topic 失效,partition 丢失,leader 失效等),此外也可以通过此参数来配置

#额外的刷新机制，默认值 600000topic.metadata.refresh.interval.ms=60000
```

2：consumer.properties

```
#消费者集群通过连接 Zookeeper 来找到 broker。

#zookeeper 连接服务器地址 zookeeper.connect=zk01:2181,zk02:2181,zk03:2181#zookeeper 的 session 过
期时间，默认 5000ms，用于检测消费者是否挂掉 zookeeper.session.timeout.ms=5000#当消费者挂掉，其他消费
者要等该指定时间才能检查到并且触发重新负载均衡 zookeeper.connection.timeout.ms=10000#这是一个时间阈
值。

#指定多久消费者更新 offset 到 zookeeper 中。

#注意 offset 更新时基于 time 而不是每次获得的消息。

#一旦在更新 zookeeper 发生异常并重启，将可能拿到已拿到过的消息 zookeeper.sync.time.ms=2000#指定消费
group.id=xxxxx



#这是一个数量阈值，经测试是 500 条。

#当 consumer 消费一定量的消息之后,将会自动向 zookeeper 提交 offset 信息#注意 offset 信息并不是每消费一次
消息就向 zk 提交

#一次,而是现在本地保存(内存),并定期提交,默认为 trueauto.commit.enable=true#  自动更新时间。默认 60 *
1000auto.commit.interval.ms=1000#  当前 consumer 的标识,可以设定,也可以有系统生成,

#主要用来跟踪消息消费情况,便于观察 conusmer.id=xxx



#  消费者客户端编号，用于区分不同客户端，默认客户端程序自动产生 client.id=xxxx
```

```
# 最大取多少块缓存到消费者(默认 10)queued.max.message.chunks=50# 当有新的 consumer 加入到 group 时,将会 reblance,此后将会

#有 partitions 的消费端迁移到新 的 consumer 上,如果一个

#consumer 获得了某个 partition 的消费权限,那么它将会向 zk

#注册 "Partition Owner registry"节点信息,但是有可能

#此时旧的 consumer 尚没有释放此节点, 此值用于控制,

#注册节点的重试次数.rebalance.max.retries=5#每拉取一批消息的最大字节数

#获取消息的最大尺寸,broker 不会像 consumer 输出大于

#此值的消息 chunk 每次 feth 将得到多条消息,此值为总大小,

#提升此值,将会消耗更多的 consumer 端内存 fetch.min.bytes=6553600#当消息的尺寸不足时,server 阻塞的时间,如果超时,

#消息将立即发送给 consumer

#数据一批一批到达，如果每一批是 10 条消息，如果某一批还

#不到 10 条，但是超时了，也会立即发送给 consumer。fetch.wait.max.ms=5000

socket.receive.buffer.bytes=655360# 如果 zookeeper 没有 offset 值或 offset 值超出范围。

#那么就给个初始的 offset。有 smallest、largest、

#anything 可选，分别表示给当前最小的 offset、

#当前最大的 offset、抛异常。默认 largestauto.offset.reset=smallest


# 指定序列化处理类 derializer.class=kafka.serializer.DefaultDecoder
```

3: service.properties

```
#broker 的全局唯一编号，不能重复 broker.id=0#用来监听链接的端口，producer 或 consumer 将在此端口建立连接 port=9092#处理网络请求的线程数量，也就是接收消息的线程数。

#接收线程会将接收到的消息放到内存中，然后再从内存中写入磁盘。num.network.threads=3#消息从内存中写入磁盘是时候使用的线程数量。
```

```
#用来处理磁盘 IO 的线程数量 num.io.threads=8#发送套接字的缓冲区大小 socket.send.buffer.bytes=102400#
接受套接字的缓冲区大小 socket.receive.buffer.bytes=102400#请求套接字的缓冲区大小
socket.request.max.bytes=104857600#kafka 运行日志存放的路径 log.dirs=/export/servers/logs/kafka



#topic 在当前 broker 上的分片个数 num.partitions=2#我们知道 segment 文件默认会被保留 7 天的时间，超时的
话就

#会被清理，那么清理这件事情就需要有一些线程来做。这里就是

#用来设置恢复和清理 data 下数据的线程数量 num.recovery.threads.per.data.dir=1#segment 文件保留的最长
时间，默认保留 7 天（168 小时），

#超时将被删除，也就是说 7 天之前的数据将被清理掉。log.retention.hours=168
```

```
log.retention.bytes
```

#滚动生成新的 segment 文件的最大时间 log.roll.hours=168 #日志文件中每个 segment 的大小，默认为 1G log.segment.bytes=1073741824 #上面的参数设置了每一个 segment 文件的大小是 1G，那么 #就需要有一个东西去定期检查 segment 文件有没有达到 1G， #多长时间去检查一次，就需要设置一个周期性检查文件大小 #的时间（单位是毫秒）。 log.retention.check.interval.ms=300000 #日志清理是否打开 log.cleaner.enable=true #broker 需要使用 zookeeper 保存 meta 数据 **zookeeper.connect=zk01:2181,zk02:2181,zk03:2181** #zookeeper 链接超时时间 zookeeper.connection.timeout.ms=6000 #上面我们说过接收线程会将接收到的消息放到内存中，然后再从内存 #写到磁盘上，那么什么时候将消息从内存中写入磁盘，就有一个 #时间限制（时间阈值）和一个数量限制（数量阈值），这里设置的是 #数量阈值，下一个参数设置的则是时间阈值。#partion buffer 中，消息的条数达到阈值，将触发 flush 到磁盘。 log.flush.interval.messages=10000 #消息 buffer 的时间，达到阈值，将触发将消息从内存 flush 到磁盘， #单位是毫秒。 log.flush.interval.ms=3000 #删除 topic 需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除 **delete.topic.enable=true** #此处的 host.name 为本机 IP(重要),如果不改,则客户端会抛出: #Producer connection to localhost:9092 unsuccessful 错误! host.name=kafka01 advertised.host.name=192.168.239.128

```
advertised.host.name=MY.EXTERNAL.IP
```

auto.create.topic.enabled = true.

default.replciation.factor =1,

4.

acks=0 模式下的运行速度是非常快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不过如果选择了这种模式， 一定会丢失一些消息

acks = 1：意味若 Leader 在收到消息并把它写入到分区数据文件

acks = all（这个和 request.required.acks = -1 含义一样）：意味着 Leader 在返回确认或错误响应之前，会等待所有同步副本都收到悄息

Producer 发送消息还可以选择同步（默认，通过 producer.type=sync 配置） 或者异步（producer.type=async）模式。如果设置成异步，虽然会极大的提高消息发送的性能，但是这样会增加丢失数据的风险。如果需要确保消息的可靠性，必须将 producer.type 设置为 sync

为了保证数据的可靠性，我们最少需要配置一下几个参数：

- producer 级别：acks=all（或者 request.required.acks=-1），同时发生模式为同步 producer.type=sync
- topic 级别：设置 replication.factor>=3，并且 min.insync.replicas>=2；
- broker 级别：关闭不完全的 Leader 选举，即 unclean.leader.election.enable=false
-

5

Consumer1 为啥消费的是 Partition0 和 Partition2，而不是 Partition0 和 Partition3？这就涉及到 Kafka 内部分区分配策略（Partition Assignment Strategy）

6

Range strategy

Range 策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。在我们的例子里面，排完序的分区将会是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9；消费者线程排完序将会是 C1-0, C2-0, C2-1。然后将 partitions 的个数除于消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区

在我们的例子里面，我们有 10 个分区，3 个消费者线程， 10 / 3 = 3，而且除不尽，那么消费者线程 C1-0 将会多消费一个分区，所以最后分区分配的结果看起来是这样的：

```
C1-0 将消费 0, 1, 2, 3 分区

C2-0 将消费 4, 5, 6 分区

C2-1 将消费 7, 8, 9 分区
```

RoundRobin strategy

使用 RoundRobin 策略有两个前提条件必须满足：

- 同一个 Consumer Group 里面的所有消费者的 num.streams 必须相等；
-

每个消费者订阅的主题必须相同。

所以这里假设前面提到的 2 个消费者的 num.streams = 2。RoundRobin 策略的工作原理：将所有主题的分区组成 TopicAndPartition 列表，然后对 TopicAndPartition 列表按照 hashCode 进行排序

7 producer API

boostStrap server, serilizer, topic name,

send can be Async, return future object,

when send is Sync, get record meta data, in future object

producerRecord(topic, paritition, timestamp, key, value), can set partition.

if no timestamp, broker will assign it.

if key is null, msg will be distribute all partition.

producer, batch size,

producer can retry

Async can not ensure order

producer.send(record, myCallback());

myCallback implement callBack {

  onComplete(RecordMetadata

}

max.in.flight.rquest.perconnnection, controler message number in flight

8. custermize partition

properties.put("partitioner.class", "sensorPartitioner");

can define logic, based on key, routing to some user defined partition.

implment partitioner(){

  public int partition()

}

9 custermized serializer, serializer object to byte[]

supplierSerializer implement serialier<Supplier> {

  byte[] serizlize(String topic, Supplier datat){

 }

}

9. consumer

'group.id", if not set group id, it means all consumer are independent and will read data from all partitions

consumers don't share partitions

suscribe , can be multiple topics, or use regular expression

consumer.poll (100) , 100 is timout mil second,

session.timeout.ms or hearbeat.interval.ms should be higher than the consumer consume time, or else, reblance may happen.

if consuemr eixt/add, group corrdinator will adjust it. manager the group of members, trigger 'rebalance'

leader(one of the consumer) will execute the rebalance activity , send new partition to cooridinator, then corrdination assgint new partition to new consumer

'rebalance' block read of all consumers

10, offset

commited offset- already processed by consumer

manaul commit : commit sync and commit aSync, consumer.commitAsync()
and consumer.commitSync();

and auto commit , config is enable.auto.commit and auto.commit.interval,ms （default 5s)

if reblance happend, and offset is not commited, but processed (because atuo commit not triggerred yet), so the msg can be process again (more than once)

11.

how to commit the processed data before a rebalance happend?

maintain a processed offset,

consumerReblanceListener, two method, onPartitionRevoked () happend before lose current partition, when consumer can commit its offset.

onPartitionAssigned() after reblancing happned.

reblanceListener.addOffset() add processed list of offset to listener. so when reblance happend, it will commit the message before reblance happend

```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
private Consumer<?,?> consumer;

public SaveOffsetsOnRebalance(Consumer<?,?> consumer) {
this.consumer = consumer;
}

public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
// save the offsets in an external store using some custom code not described here
for(TopicPartition partition: partitions)
saveOffsetInExternalStore(consumer.position(partition));
}
```

12. partition assign control

TopicPartition p0 = new TopicPartition(topicname , 0);

TopicPartition p1 = new TopicPartition(topicname , 1);

//assign partition to consumer manually

consumer.assign(Arrays.asList(p0,p1));

consumer.seek(p0, 22) -> set offset manually

13. schema

include schema with data

benefit is that can change schema without changing code.

avro 1. define a schema, 2 serialize code based on schema 3.deserialize based on schema

KafkaAvroSerializer

Schema regestry, KafkaAvroSerializer use the schema stored in schema registry, in message, only has schema id (so reduce msg size)

compatibility type: BACKWARD , FORWARD(last version)

new fix threadpool , using blocking queue,

newFixedThreadPool 只有核心线程，并且不存在超时机制，采用 LinkedBlockingQueue，所以对于任务队列的大小也是没有限制的 , use linkedBlockign queue, unbounded

new cache pool, if thread core is full, new thread is put in pool , will be killed after 60 second. use Synchonous queue. 内部没有任何容量的阻塞队列。在它内部没有任何的缓存空间。对于 SynchronousQueue 中的数据元素只有当我们试着取走的时候才可能存在

schedule pool,

newSingleThreadExecutor 保证线程一个一个的执行。

ThreadPoolExecutor 参数含义

1. corePoolSize 线程池中的核心线程数，默认情况下，核心线程一直存活在线程池中，即便他们在线程池中处于闲置状态。除非我们将 ThreadPoolExecutor 的 allowCoreThreadTimeOut 属性设为 true 的时候，这时候处于闲置的核心线程在等待新任务到来时会有超时策略，这个超时时间由

keepAliveTime 来指定。一旦超过所设置的超时时间，闲置的核心线程就会被终止。**2. maximumPoolSize** 线程池中所容纳的最大线程数，如果活动的线程达到这个数值以后，后续的新任务将会被阻塞。**3. keepAliveTime** 非核心线程闲置时的超时时长，对于非核心线程，闲置时间超过这个时间，非核心线程就会被回收。只有对 ThreadPoolExecutor 的 allowCoreThreadTimeOut 属性设为 true 的时候，这个超时时间才会对核心线程产生效果。**4. unit** 用于指定 keepAliveTime 参数的时间单位。他是一个枚举，可以使用的单位有天（TimeUnit.DAYS），小时（TimeUnit.HOURS），分钟（TimeUnit.MINUTES），毫秒(TimeUnit.MILLISECONDS)，微秒(TimeUnit.MICROSECONDS, 千分之一毫秒)和毫微秒(TimeUnit.NANOSECONDS, 千分之一微秒);**5. workQueue** 线程池中保存等待执行的任务的阻塞队列。通过线程池中的 execute 方法提交的 Runable 对象都会存储在该队列中。我们可以选择下面几个阻塞队列。

- **ArrayBlockingQueue:基于数组实现的有界的阻塞队列,该队列按照 FIFO（先进先出）原则对队列中的元素进行排序。**
- LinkedBlockingQueue:基于链表实现的阻塞队列，该队列按照 FIFO（先进先出）原则对队列中的元素进行排序。
- SynchronousQueue:内部没有任何容量的阻塞队列。在它内部没有任何的缓存空间。对于 SynchronousQueue 中的数据元素只有当我们试着取走的时候才可能存在。
- PriorityBlockingQueue:具有优先级的无限阻塞队列。
- 我们还能够通过实现 BlockingQueue 接口来自定义我们所需要的阻塞队列

6. threadFactory 线程工厂，为线程池提供新线程的创建。ThreadFactory 是一个接口，里面只有一个 newThread 方法。**7. handler** 他是 RejectedExecutionHandler 对象，而 RejectedExecutionHandler 是一个接口，里面只有一个 rejectedExecution 方法。当任务队列已满并且线程池中的活动线程已经达到所限定的最大值或者是无法成功执行任务，这时候 ThreadPoolExecutor 会调用 RejectedExecutionHandler 中的 rejectedExecution 方法。在 ThreadPoolExecutor 中有四个内部类实现了 RejectedExecutionHandler 接口。在线程池中它默认是 AbortPolicy，在无法处理新任务时抛出 RejectedExecutionException 异常。下面是在 ThreadPoolExecutor 中提供的四个可选值。

- **CallerRunsPolicy:如果满了， 改成只用调用者所在线程来运行任务， 即 main thread 来执行当前线程**
- AbortPolicy:直接抛出 RejectedExecutionException 异常。
- DiscardPolicy:丢弃掉该任务，不进行处理
- DiscardOldestPolicy:丢弃队列里最近的一个任务，并执行当前任务。
- **我们也可以通过实现 RejectedExecutionHandler 接口来自定义我们自己的 handler。如记录日志或持久化不能处理的任务。**

ThreadPoolExecutor 执行规则

1. 如果在线程池中的线程数量没有达到核心的线程数量，这时候就回启动一个核心线程来执行任务。
2. 如果线程池中的线程数量已经超过核心线程数，这时候任务就会被插入到任务队列中排队等待执行。
3. 由于任务队列已满，无法将任务插入到任务队列中。这个时候如果线程池中的线程数量没有达到线程池所设定的最大值，那么这时候就会立即启动一个非核心线程来执行任务。
4. 如果线程池中的数量达到了所规定的最大值，那么就会拒绝执行此任务，这时候就会调用 RejectedExecutionHandler 中的 rejectedExecution 方法来通知调用者。

```
ExecutorService executorService = new ThreadPoolExecutor(5,10,10,TimeUnit.MILLISECONDS,new LinkedBlockingQueue<>());
```

execute

当我们使用 execute 来提交任务时，由于 execute 方法没有返回值，所以说我们也就无法判定任务是否被线程池执行成功。

submit

当我们使用 submit 来提交任务时,它会返回一个 future,我们就可以通过这个 future 来判断任务是否执行成功，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用 get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时候有可能任务并没有执行完。

JVM

happend before rule apply to both 'volatile' and 'sychronized' key word, will ensure the value in shared memeory is the latest.

servlet can use Async context.

AsyncContext =request.startAsync()

condition signal 唤醒 waiting thread， 唤醒一个等待时机最长的。

condition await will **release** it automatically

如果 IO wait 时间太长， 使用 callback

A Flux is the equivalent of an RxJava

或者 java fiber 轻量级的接口， thread 占用内存少

lock。getHoldCount 查看 lock call 了多少次

new reentrantLock(true) 是公平锁

trylock， return true means get the lock

readwriteLock, one write lock, multiple read lock

readlock and write lock are seperate instance, only one is allowed at a time.

thread interruct, but call interrupt method

blocking thread: object wait, thread.sleep, thread.join

semphore , 1) acquire permit to call 2) then release ()

new Semaphore(3) -> only three thread is allowed to call

tryAcquire(), tryAcquire(timeout)

make thread timeout, 1) stop the thread by shutdown the threadpool, shutdownNow() will return all thread that is not excuted. 2) future cancel the callable. 3) interrupt thread 4) while a volatile variable true to stop 5) schedule service

consumer/producer model implmentation by 'blockingQueue' ArrayBlockingQueue

how to wait N task , with timeout

CountdownLatch latch = new Countdwonlatch(3);

latch.await(3, timeUnit.Second), //wait 3 second or countdown to 0, what ever come first.

forkjoin pool,

create sub task, then join together ,

each thread has its own queue , called 'deque' (store sub task)

atomic long synchronized only happedn during 'sum

stiped lock 实现 lock 自动分组，， 比如十个对象共享一个 Lock

Sticked.lock(10);

lockid= hash(object) %locknumber

avoid decklock: 1) timeout 2) order of the lock

```
//get the first 3 element

IntStream.of(numbers).distinct().sorted().limit(3)
```