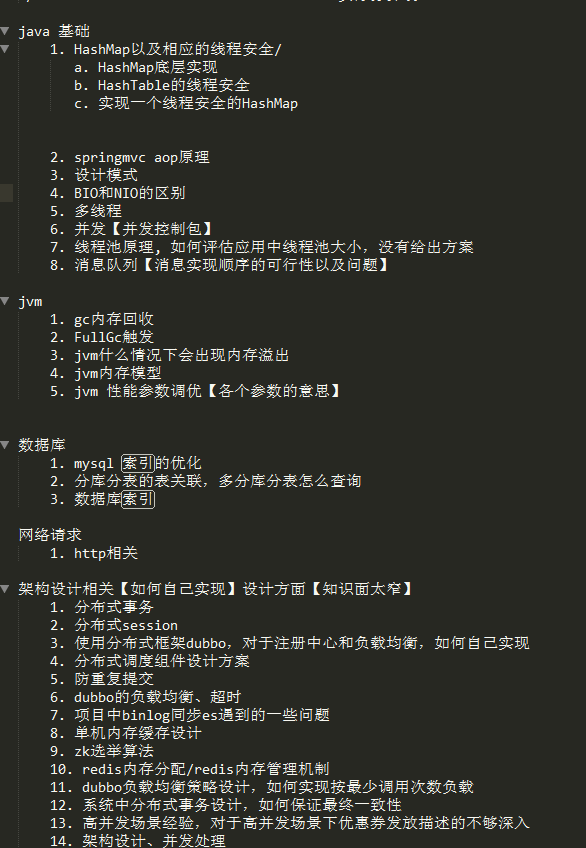
# Java面试题集



## Java基础

### 1.HashMap实现原理

1、基于哈希表（数组+链表+二叉树（**红黑树**））JDK1.8

2、默认加载因子为0.75，默认数组大小是16（1 << 4

）

3、把对象存储到哈希表中，如何存储？

把key对象通过hash()方法计算hash值，然后用这个hash值对数组长度取余数（默认16），来决定对该key的对象在数组中的位置index，当这个位置有多个对象时，以链表结构存储，JDK1.8后，当链表长度大于8时，链表将转换为红黑树结构存储。（这样的目的是为了取值更快，存储数据量越大，性能表现越明显。）

4、扩容原理：当数组的容量超过了负载因子，你们表示该数组需要扩容，如何扩容？

扩充算法是：当前数组容量的<<1（相对应乘2），扩大一倍，扩容次数过多，会影响性能，每次扩容表示哈希表重新散列（重新计算每个对象的存储位置），我们在开发中尽量要减少扩充次数带来的性能问题。

5、线程不安全，适合在单线程中使用。

# Java基础及强化

## ※Java API java.io java.nio java.lang java.util

## ※Java 8新新特性

### 一. 流式处理简介

在我接触到java8流式处理的时候，我的第一感觉是流式处理让集合操作变得简洁了许多，通常我们需要多行代码才能完成的操作，借助于流式处理可以在一行中实现。比如我们希望对一个包含整数的集合中筛选出所有的偶数，并将其封装成为一个新的List返回，那么在java8之前，我们需要通过如下代码实现：

List<Integer> evens = new ArrayList<>();

for (final Integer num : nums) {

if (num % 2 == 0) {

evens.add(num);

}

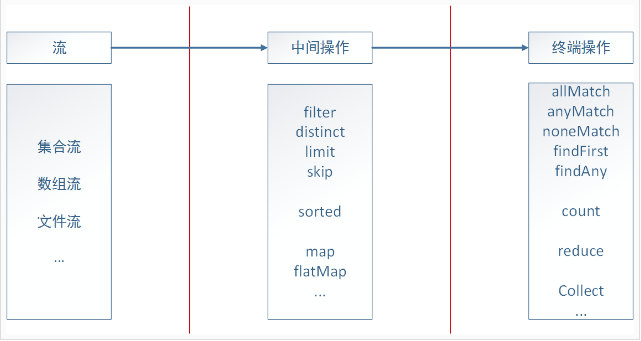
}

通过java8的流式处理，我们可以将代码简化为：

List<Integer> evens = nums.stream().filter(num -> num % 2 == 0).collect(Collectors.toList());

先简单解释一下上面这行语句的含义，stream()操作将集合转换成一个流，filter()执行我们自定义的筛选处理，这里是通过lambda表达式筛选出所有偶数，最后我们通过collect()对结果进行封装处理，并通过Collectors.toList()指定其封装成为一个List集合返回。

由上面的例子可以看出，java8的流式处理极大的简化了对于集合的操作，实际上不光是集合，包括数组、文件等，只要是可以转换成流，我们都可以借助流式处理，类似于我们写SQL语句一样对其进行操作。java8通过内部迭代来实现对流的处理，一个流式处理可以分为三个部分：转换成流、中间操作、终端操作。如下图：



以集合为例，一个流式处理的操作我们首先需要调用stream()函数将其转换成流，然后再调用相应的中间操作达到我们需要对集合进行的操作，比如筛选、转换等，最后通过终端操作对前面的结果进行封装，返回我们需要的形式。

### 二. 中间操作

我们定义一个简单的学生实体类，用于后面的例子演示：

**public** **class** Student {

/\*\* 学号 \*/

**private** **long** id;

**private** String name;

**private** **int** age;

/\*\* 年级 \*/

**private** **int** grade;

/\*\* 专业 \*/

**private** String major;

/\*\* 学校 \*/

**private** String school;

**package** com.xgb.org.stream;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** MainStreamTest {

**public** **static** **void** main(String[] args) {

List<Student> students = *init*();

students = students.stream().filter(student->"武汉大学".equals(student.getSchool())).sorted((s1 ,s2) -> s1.getAge() - s2.getAge() ).skip(2).collect(Collectors.*toList*());

students.forEach( (student1) -> { System.***out***.println("student: " + student1);} );

// List<Integer> ints = initInt();

// ints = ints.stream().filter((intList) -> intList % 2 == 0).distinct().limit(10).collect(Collectors.toList());

// ints.forEach( (intList) -> { System.out.println("intList: " + intList);} );

}

**public** **static** List<Student> init() {

List<Student> list = **new** ArrayList<Student>();

list.add(**new** Student(20160001, "孔明", 20, 1, "土木工程", "武汉大学"));

list.add(**new** Student(20160002, "伯约", 21, 2, "信息安全", "武汉大学"));

list.add(**new** Student(20160003, "玄德", 22, 3, "经济管理", "武汉大学"));

list.add(**new** Student(20160004, "云长", 21, 2, "信息安全", "武汉大学"));

list.add(**new** Student(20161001, "翼德", 21, 2, "机械与自动化", "华中科技大学"));

list.add(**new** Student(20161002, "元直", 23, 4, "土木工程", "华中科技大学"));

list.add(**new** Student(20161003, "奉孝", 23, 4, "计算机科学", "华中科技大学"));

list.add(**new** Student(20162001, "仲谋", 22, 3, "土木工程", "浙江大学"));

list.add(**new** Student(20162002, "鲁肃", 23, 4, "计算机科学", "浙江大学"));

list.add(**new** Student(20163001, "丁奉", 24, 5, "土木工程", "南京大学"));

**return** list;

}

**public** **static** List<Integer> initInt() {

Integer[] arr = { 1,4,7,4,9,2,4,5,66,12,43 };

List<Integer> list = **new** ArrayList<Integer>(Arrays.*asList*(arr));

**return** list;

}

}

#### 2.1 过滤

过滤，顾名思义就是按照给定的要求对集合进行筛选满足条件的元素，java8提供的筛选操作包括：filter、distinct、limit、skip。

**filter**  
在前面的例子中我们已经演示了如何使用filter，其定义为：Stream<T> filter(Predicate<? super T> predicate)，filter接受一个谓词Predicate，我们可以通过这个谓词定义筛选条件，在介绍lambda表达式时我们介绍过Predicate是一个函数式接口，其包含一个test(T t)方法，该方法返回boolean。现在我们希望从集合students中筛选出所有武汉大学的学生，那么我们可以通过filter来实现，并将筛选操作作为参数传递给filter：

List<Student> whuStudents = students.stream()

.filter(student -> "武汉大学".equals(student.getSchool()))

.collect(Collectors.toList());

**distinct**  
distinct操作类似于我们在写SQL语句时，添加的DISTINCT关键字，用于去重处理，distinct基于Object.equals(Object)实现，回到最开始的例子，假设我们希望筛选出所有不重复的偶数，那么可以添加distinct操作：

List<Integer> evens = nums.stream()

.filter(num -> num % 2 == 0).distinct()

.collect(Collectors.toList());

**limit**  
limit操作也类似于SQL语句中的LIMIT关键字，不过相对功能较弱，limit返回包含前n个元素的流，当集合大小小于n时，则返回实际长度，比如下面的例子返回前两个专业为土木工程专业的学生：

List<Student> civilStudents = students.stream()

.filter(student -> "土木工程".equals(student.getMajor())).limit(2)

.collect(Collectors.toList());

说到limit，不得不提及一下另外一个流操作：sorted。该操作用于对流中元素进行排序，sorted要求待比较的元素必须实现Comparable接口，如果没有实现也不要紧，我们可以将比较器作为参数传递给sorted(Comparator<? super T> comparator)，比如我们希望筛选出专业为土木工程的学生，并按年龄从小到大排序，筛选出年龄最小的两个学生，那么可以实现为：

List<Student> sortedCivilStudents = students.stream()

.filter(student -> "土木工程".equals(student.getMajor())).sorted((s1, s2) -> s1.getAge() - s2.getAge())

.limit(2)

.collect(Collectors.toList());

**skip**  
skip操作与limit操作相反，如同其字面意思一样，是跳过前n个元素，比如我们希望找出排序在2之后的土木工程专业的学生，那么可以实现为：

List<Student> civilStudents = students.stream()

.filter(student -> "土木工程".equals(student.getMajor()))

.skip(2)

.collect(Collectors.toList());

通过skip，就会跳过前面两个元素，返回由后面所有元素构造的流，如果n大于满足条件的集合的长度，则会返回一个空的集合。

#### 2.2 映射

在SQL中，借助SELECT关键字后面添加需要的字段名称，可以仅输出我们需要的字段数据，而流式处理的映射操作也是实现这一目的，在java8的流式处理中，主要包含两类映射操作：map和flatMap。

**map**  
举例说明，假设我们希望筛选出所有专业为计算机科学的学生姓名，那么我们可以在filter筛选的基础之上，通过map将学生实体映射成为学生姓名字符串，具体实现如下：

List<String> names = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.map(Student::getName).collect(Collectors.toList());

除了上面这类基础的map，java8还提供了mapToDouble(ToDoubleFunction<? super T> mapper)，mapToInt(ToIntFunction<? super T> mapper)，mapToLong(ToLongFunction<? super T> mapper)，这些映射分别返回对应类型的流，java8为这些流设定了一些特殊的操作，比如我们希望计算所有专业为计算机科学学生的年龄之和，那么我们可以实现如下：

int totalAge = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.mapToInt(Student::getAge).sum();

通过将Student按照年龄直接映射为IntStream，我们可以直接调用提供的sum()方法来达到目的，此外使用这些数值流的好处还在于可以避免jvm装箱操作所带来的性能消耗。

**flatMap**  
flatMap与map的区别在于 **flatMap是将一个流中的每个值都转成一个个流，然后再将这些流扁平化成为一个流** 。举例说明，假设我们有一个字符串数组String[] strs = {"java8", "is", "easy", "to", "use"};，我们希望输出构成这一数组的所有非重复字符，那么我们可能首先会想到如下实现：

List<String[]> distinctStrs = Arrays.stream(strs)

.map(str -> str.split("")) // 映射成为Stream<String[]>

.distinct()

.collect(Collectors.toList());

在执行map操作以后，我们得到是一个包含多个字符串（构成一个字符串的字符数组）的流，此时执行distinct操作是基于在这些字符串数组之间的对比，所以达不到我们希望的目的，此时的输出为：

[j, a, v, a, 8]

[i, s]

[e, a, s, y]

[t, o]

[u, s, e]

distinct只有对于一个包含多个字符的流进行操作才能达到我们的目的，即对Stream<String>进行操作。此时flatMap就可以达到我们的目的：

List<String> distinctStrs = Arrays.stream(strs)

.map(str -> str.split("")) // 映射成为Stream<String[]>

.flatMap(Arrays::stream) // 扁平化为Stream<String>

.distinct()

.collect(Collectors.toList());

flatMap将由map映射得到的Stream<String[]>，转换成由各个字符串数组映射成的流Stream<String>，再将这些小的流扁平化成为一个由所有字符串构成的大流Steam<String>，从而能够达到我们的目的。  
与map类似，flatMap也提供了针对特定类型的映射操作：flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)，flatMapToInt(Function<? super T,? extends IntStream> mapper)，flatMapToLong(Function<? super T,? extends LongStream> mapper)。

### 三. 终端操作

终端操作是流式处理的最后一步，我们可以在终端操作中实现对流查找、归约等操作。

#### 3.1 查找

**allMatch**  
allMatch用于检测是否全部都满足指定的参数行为，如果全部满足则返回true，例如我们希望检测是否所有的学生都已满18周岁，那么可以实现为：

boolean isAdult = students.stream().allMatch(student -> student.getAge() >= 18);

**anyMatch**  
anyMatch则是检测是否存在一个或多个满足指定的参数行为，如果满足则返回true，例如我们希望检测是否有来自武汉大学的学生，那么可以实现为：

boolean hasWhu = students.stream().anyMatch(student -> "武汉大学".equals(student.getSchool()));

**noneMathch**  
noneMatch用于检测是否不存在满足指定行为的元素，如果不存在则返回true，例如我们希望检测是否不存在专业为计算机科学的学生，可以实现如下：

boolean noneCs = students.stream().noneMatch(student -> "计算机科学".equals(student.getMajor()));

**findFirst**  
findFirst用于返回满足条件的第一个元素，比如我们希望选出专业为土木工程的排在第一个学生，那么可以实现如下：

Optional<Student> optStu = students.stream().filter(student -> "土木工程".equals(student.getMajor())).findFirst();

findFirst不携带参数，具体的查找条件可以通过filter设置，此外我们可以发现findFirst返回的是一个Optional类型，关于该类型的具体讲解可以参考上一篇：[Java8新特性 – Optional类](http://www.codeceo.com/article/optional-class-of-java-8.html)。

**findAny**  
findAny相对于findFirst的区别在于，findAny不一定返回第一个，而是返回任意一个，比如我们希望返回任意一个专业为土木工程的学生，可以实现如下：

Optional<Student> optStu = students.stream().filter(student -> "土木工程".equals(student.getMajor())).findAny();

实际上对于顺序流式处理而言，findFirst和findAny返回的结果是一样的，至于为什么会这样设计，是因为在下一篇我们介绍的**并行流式处理**，当我们启用并行流式处理的时候，查找第一个元素往往会有很多限制，如果不是特别需求，在并行流式处理中使用findAny的性能要比findFirst好。

#### 3.2 归约

前面的例子中我们大部分都是通过collect(Collectors.toList())对数据封装返回，如我的目标不是返回一个新的集合，而是希望对经过参数化操作后的集合进行进一步的运算，那么我们可用对集合实施归约操作。java8的流式处理提供了reduce方法来达到这一目的。

前面我们通过mapToInt将Stream<Student>映射成为IntStream，并通过IntStream的sum方法求得所有学生的年龄之和，实际上我们通过归约操作，也可以达到这一目的，实现如下：

[复制代码](javascript:void(0);)

// 前面例子中的方法

int totalAge = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.mapToInt(Student::getAge).sum();

// 归约操作

int totalAge = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.map(Student::getAge)

.reduce(0, (a, b) -> a + b);

// 进一步简化

int totalAge2 = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.map(Student::getAge)

.reduce(0, Integer::sum);

// 采用无初始值的重载版本，需要注意返回Optional

Optional<Integer> totalAge = students.stream()

.filter(student -> "计算机科学".equals(student.getMajor()))

.map(Student::getAge)

.reduce(Integer::sum); // 去掉初始值

[复制代码](javascript:void(0);)

#### 3.3 收集

前面利用collect(Collectors.toList())是一个简单的收集操作，是对处理结果的封装，对应的还有toSet、toMap，以满足我们对于结果组织的需求。这些方法均来自于java.util.stream.Collectors，我们可以称之为收集器。

##### 3.3.1 归约

收集器也提供了相应的归约操作，但是与reduce在内部实现上是有区别的，收集器更加适用于可变容器上的归约操作，这些收集器广义上均基于Collectors.reducing()实现。

例1：求学生的总人数

long count = students.stream().collect(Collectors.counting());

// 进一步简化

long count = students.stream().count();

例2：求年龄的最大值和最小值

[复制代码](javascript:void(0);)

// 求最大年龄

Optional<Student> olderStudent = students.stream().collect(Collectors.maxBy((s1, s2) -> s1.getAge() - s2.getAge()));

// 进一步简化

Optional<Student> olderStudent2 = students.stream().collect(Collectors.maxBy(Comparator.comparing(Student::getAge)));

// 求最小年龄

Optional<Student> olderStudent3 = students.stream().collect(Collectors.minBy(Comparator.comparing(Student::getAge)));

[复制代码](javascript:void(0);)

例3：求年龄总和

int totalAge4 = students.stream().collect(Collectors.summingInt(Student::getAge));

对应的还有summingLong、summingDouble。

例4：求年龄的平均值

double avgAge = students.stream().collect(Collectors.averagingInt(Student::getAge));

对应的还有averagingLong、averagingDouble。

例5：一次性得到元素个数、总和、均值、最大值、最小值

IntSummaryStatistics statistics = students.stream().collect(Collectors.summarizingInt(Student::getAge));

输出：

IntSummaryStatistics{count=10, sum=220, min=20, average=22.000000, max=24}

对应的还有summarizingLong、summarizingDouble。

例6：字符串拼接

String names = students.stream().map(Student::getName).collect(Collectors.joining());

// 输出：孔明伯约玄德云长翼德元直奉孝仲谋鲁肃丁奉

String names = students.stream().map(Student::getName).collect(Collectors.joining(", "));

// 输出：孔明, 伯约, 玄德, 云长, 翼德, 元直, 奉孝, 仲谋, 鲁肃, 丁奉

##### 3.3.2 分组

在数据库操作中，我们可以通过GROUP BY关键字对查询到的数据进行分组，java8的流式处理也为我们提供了这样的功能Collectors.groupingBy来操作集合。比如我们可以按学校对上面的学生进行分组：

Map<String, List<Student>> groups = students.stream().collect(Collectors.groupingBy(Student::getSchool));

groupingBy接收一个分类器Function<? super T, ? extends K> classifier，我们可以自定义分类器来实现需要的分类效果。

上面演示的是一级分组，我们还可以定义多个分类器实现 **多级分组**，比如我们希望在按学校分组的基础之上再按照专业进行分组，实现如下：

Map<String, Map<String, List<Student>>> groups2 = students.stream().collect(

Collectors.groupingBy(Student::getSchool, // 一级分组，按学校

Collectors.groupingBy(Student::getMajor))); // 二级分组，按专业

实际上在groupingBy的第二个参数不是只能传递groupingBy，还可以传递任意Collector类型，比如我们可以传递一个Collector.counting，用以统计每个组的个数：

Map<String, Long> groups = students.stream().collect(Collectors.groupingBy(Student::getSchool, Collectors.counting()));

如果我们不添加第二个参数，则编译器会默认帮我们添加一个Collectors.toList()。

##### 3.3.3 分区

分区可以看做是分组的一种特殊情况，在分区中key只有两种情况：true或false，目的是将待分区集合按照条件一分为二，java8的流式处理利用ollectors.partitioningBy()方法实现分区，该方法接收一个谓词，例如我们希望将学生分为武大学生和非武大学生，那么可以实现如下：

Map<Boolean, List<Student>> partition = students.stream().collect(Collectors.partitioningBy(student -> "武汉大学".equals(student.getSchool())));

分区相对分组的优势在于，我们可以同时得到两类结果，在一些应用场景下可以一步得到我们需要的所有结果，比如将数组分为奇数和偶数。

以上介绍的所有收集器均实现自接口java.util.stream.Collector，该接口的定义如下：

[复制代码](javascript:void(0);)

public interface Collector<T, A, R> {

/\*\*

\* A function that creates and returns a new mutable result container.

\*

\* @return a function which returns a new, mutable result container

\*/

Supplier<A> supplier();

/\*\*

\* A function that folds a value into a mutable result container.

\*

\* @return a function which folds a value into a mutable result container

\*/

BiConsumer<A, T> accumulator();

/\*\*

\* A function that accepts two partial results and merges them. The

\* combiner function may fold state from one argument into the other and

\* return that, or may return a new result container.

\*

\* @return a function which combines two partial results into a combined

\* result

\*/

BinaryOperator<A> combiner();

/\*\*

\* Perform the final transformation from the intermediate accumulation type

\* {@code A} to the final result type {@code R}.

\*

\* <p>If the characteristic {@code IDENTITY\_TRANSFORM} is

\* set, this function may be presumed to be an identity transform with an

\* unchecked cast from {@code A} to {@code R}.

\*

\* @return a function which transforms the intermediate result to the final

\* result

\*/

Function<A, R> finisher();

/\*\*

\* Returns a {@code Set} of {@code Collector.Characteristics} indicating

\* the characteristics of this Collector. This set should be immutable.

\*

\* @return an immutable set of collector characteristics

\*/

Set<Characteristics> characteristics();

}

[复制代码](javascript:void(0);)

我们也可以实现该接口来定义自己的收集器，此处不再展开。

### 四. 并行流式数据处理

## Java虚拟机

### JVM的组成

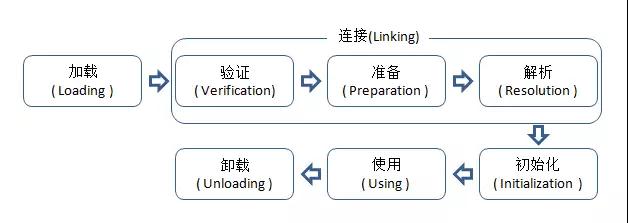
#### （1）、类加载器

在JVM启动时或者类运行时将需要的class加载到JVM中。

##### 问：Java 虚拟机是如何判定两个 Java 类是相同的？

答：Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA和 ClassLoaderB分别读取了这个 Sample.class文件，并定义出两个 java.lang.Class类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException。

##### 类加载器死如何加载class文件的？



第一个阶段是找到 .class 文件并把这个文件包含的字节码加载到内存中

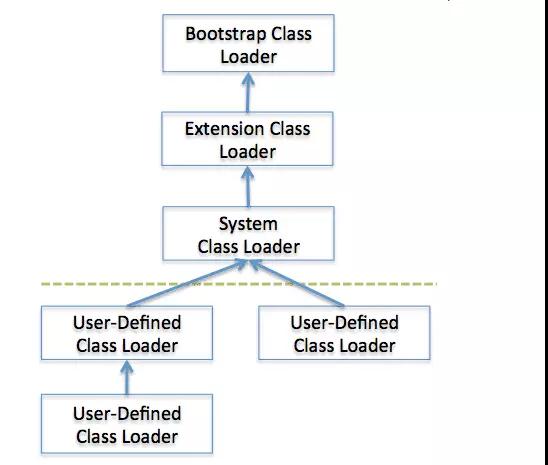
第二阶段又可以分为三个步骤，分别是**字节码验证**、**Class 类数据结构分析**及**相应的内存分配**和最后的**符号表的链接**

第三个阶段是类中静态属性和初始化赋值，以及静态块的执行等

##### 双亲委派模型（Parent Delegation Modle）

解析：类的加载过程采用双亲委派机制，这种机制能更好的保证 Java 平台的安全性

答：类加载器 ClassLoader 是具有层次结构的，也就是父子关系，其中，Bootstrap 是所有类加载器的父亲，如下图所示：



该模型要求除了顶层的 Bootstrap class loader 启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。每个类加载器都有自己的命名空间（由该加载器及所有父类加载器所加载的类组成，在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类）

##### 问：双亲委派模型的工作过程？

先查询当前的ClassLord此类是否已经加载，如果加载了就直接返回，如果没有加载就委托父加载器，父加载器使用相同的策略，首先查看自己的缓存，然后委托父类去加载，一直到BootStrap ClassLorder。当所有的父类加载器都没有加载的时候，再有当前的类加载器加载，并将其放入到自己的缓存中，以便下次有加载请求的时候直接返回。

##### 问：问什么使用这样的手设计呢？/使用泽中模型来组织的好处。

答：主要是为了安全性，避免用户自己编写的类动态替换Java的一些核心类，比如String，同时也避免了重复加载，相同的class文件被不同的ClassLord加载就是不同的两个类，如果相互转换类型的话就会抛出java.lang.ClassCaseException.

#### （2）、执行引擎

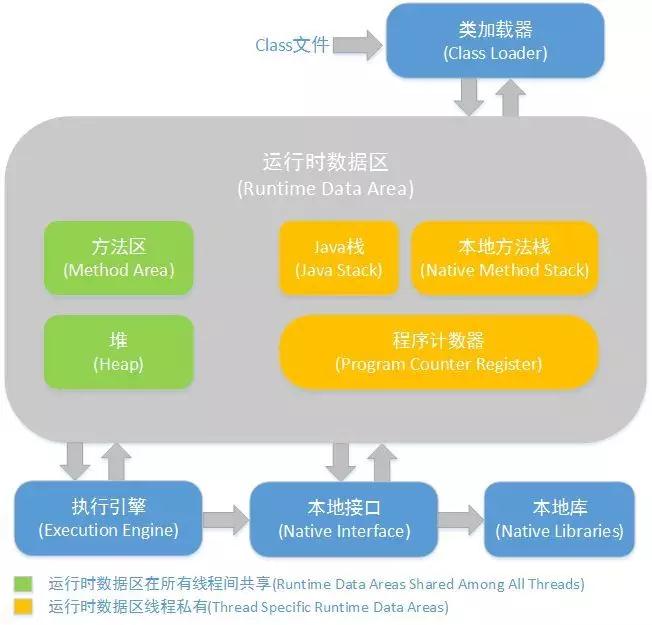
执行引擎的任务是负责执行class文件中包含的字节码指令，相对实际机器上的CPU。

#### （3）、内存区

将内存划分成若干个区以模拟实际机器上的存储、记录和调度，如实际机器上的各种功能的寄存器或者PC指针的记录等。

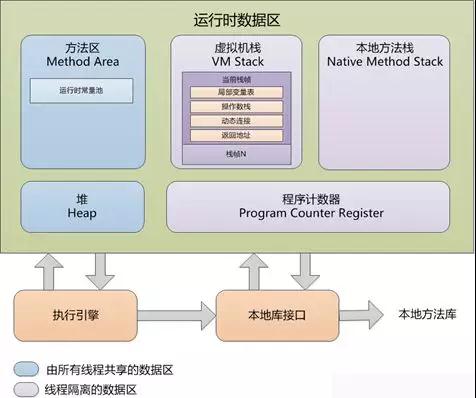
#### （4）、本地方法调用

调用C或C++实现的本地方法的代码返回结果。



### JVM 内存管理

JVM内存划分：



#### 1.线程计数器

线程计数器，是一块较小的内存空间，用来指定当前线程执行字节码的行数，每个线程计数器都是私有的，因为每个线程都需要记录执行的行数；这里解释一下为什么每个线程都需要一个线程计数器，JVM的多线程是通过线程轮流切换分配执行时间来实现的，在任何时刻，每个处理器都只会执行一个线程中的指令，当线程进行切换的时，为了线程能恢复当正确的位置，所以每个线程必须有个独立的线程计数器，这样才能保证线程之间不互相影响。

　　这里注意下，如果线程执行是一个Java方法的时候，计数器记录的是虚拟机字节码指令的地址；当执行的是Native的方法的时候，计数器指令为空；该内存区域是Java虚拟机唯一没有规定任何OutOfMemoryError的区域。

#### 2.Java虚拟机栈

Java虚拟栈，这个也是一个线程私有的，生命周期与线程是同步的，每个方法在执行的同时，都会创建一个栈帧，用于存储局部变量表，操作数栈，动态链接，方法出入口等信息，每个方法的调用到执行完成的过程就是一个栈帧入栈到出栈的过程；

　　这里解释一下局部变量表，局部变量表存储方法相关的局部变量，包括基本数据，对象引用和返回地址等。在局部变量表中，只有long和double类型会占用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变；

虚拟机栈规定了２种异常情况，一种是线程请求栈的深度大于虚拟机栈所允许的深度，这时候将会抛出StackOverflowError异常，如果当Java虚拟机允许动态扩展虚拟机栈的时候，当扩展的时候没办法分配到内存的时候就会报OutOfMemoryError异常；

#### 3.本地方法栈

与虚拟机栈执行的基本相同，唯一的区别就是虚拟机栈是执行Java方法的，本地方法栈是执行native方法的；

#### 4.Java堆

Java堆，堆区是Java虚拟机所管理的内存中最大的一块，Java堆是被所有线程共享的内存区域，主要存储对象的实例。

       当堆中没有内存完成实例分配，并且堆无法扩展的时候，将会抛出OutOfMemoryError异常；当前虚拟机都是可以扩展的；

#### 5.方法区

方法区，这个也是线程共享的内存区域，存储被虚拟机加载的类信息、常量、静态变量、即时编译的代码数据等；

      方法区在物理上也是不需要连续的，可以选择固定大小或者扩展的大小，还可以选择不实现垃圾收集，方法区的垃圾回收是比较少的，这就是方法区为什么被称为永久区的原因，但是方法区也是可以执行回收的，该区域主要是针对常量池和类型的卸载；在方法区也规定当方法区无法满足内存分布的时候，将会抛出OutOfMemoryError异常；

      运行时常量是方法区的一部分，常量池主要用于存放编译生成的各种字面量和符合引用，由于常量池属于方法区的一部分，所以当常量池没有内存空间的时候就抛出OutOfMemoryError异常；

#### 6.直接内存

直接内存，不是虚拟机运行时的一部分，可以直接访问堆外的内存；所以当内存空间无法动态扩展的时候就会出现OutOfMemoryError异常；

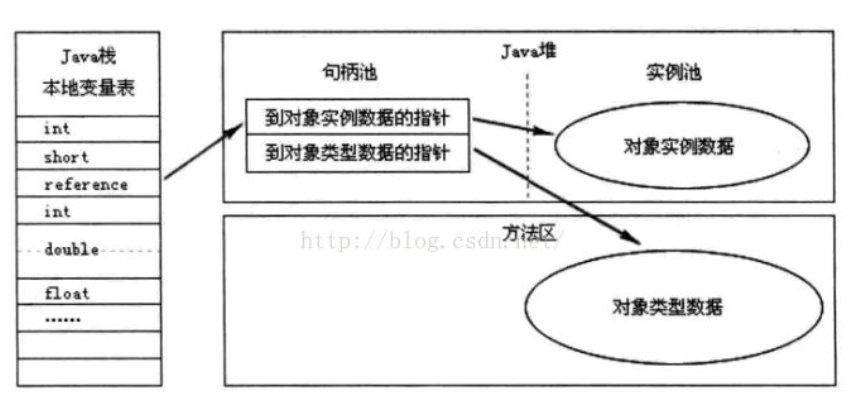
### 总述

 以上基本是JVM内存分布的内容，简单的理解水满则溢出就是这个道理，系统的整个空间是一个大的容器，分不同的部分或者桶去分担整个容量，当那个桶不够的时候自然会溢出。明白内存区域的分布我们看下对象是如何分配在内存空间里面的?

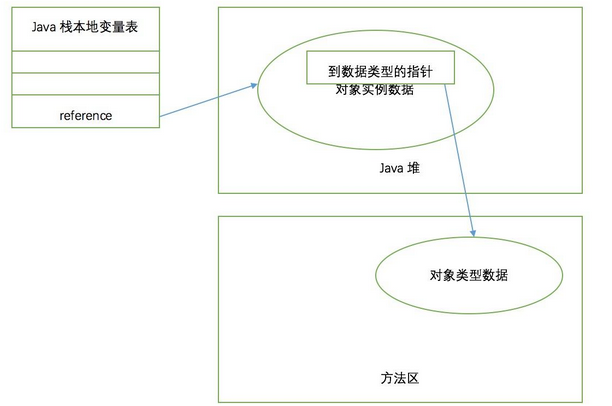
  Java对象这里指的是引用类型的对象，这里用Student stu=new Student()为例子访问，Student stu作为引用对象，存在与Java虚拟机栈上，new Student()保存在Java堆中，堆中记录Student类型的信息包括方法，接口，对象类型等地址，这些类型的执行的数据存储在方法区中；

  这里需要说明一下对象访问的方式，主要包括2种句柄访问和直接指针访问：

##### 1. 句柄访问主要是Java堆中划分一块句柄池，虚拟机栈中存放句柄池中的地址，句柄池中包括对象的实例数据和对象类型的数据的地址，基本分布如下图：



##### 2.直接指针访问，就是虚拟机栈直接指向Java堆中的对象类型指针和对象的实例数据，然后对象类型指针在指向方法区中对象类型的实例数据，分布如下图：



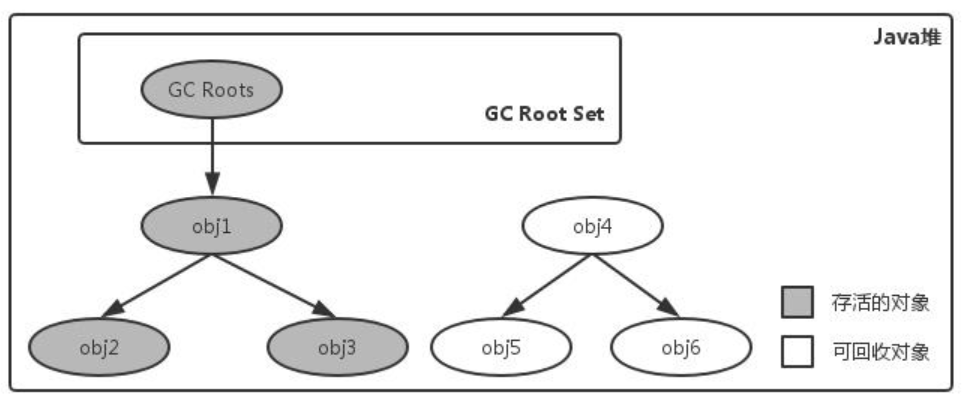
### GC回收

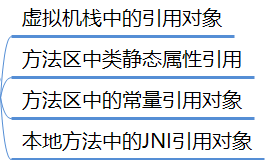
HotSpot就是第二种访问方式，优点在于访问速度快，省去一次指针开销时间，JVM内存分布基本介绍到这里，接下来说下如何保证正确回收？

  回收是已经没有用的对象，那怎么判断一个对象没用引用？这里需要简单介绍2种方法:引用计数法和可达性分析算法；

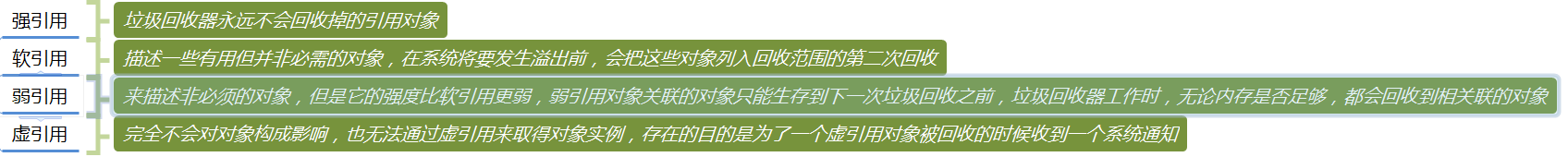
这里简单说一下引用计数法：对象中添加一个引用计数器，每当有一个地方引用计数器就增加1，引用失效就减少1，计数器为0就不可用；缺点就在于无法处理对象直接相互引用的问题，因为相互引用以后无法使计数器为0，所以无法回收；

  可达性分析算法，也就是我们常说的GC Root,，当一个对象没有与任何引用链相连的时候，就可以对该对象进行回收，下面是Java中GC Root对象使用的几个地方：





   以上对象简单就是分为可用和不可用这2种，现在Java对引用概念进行扩充：



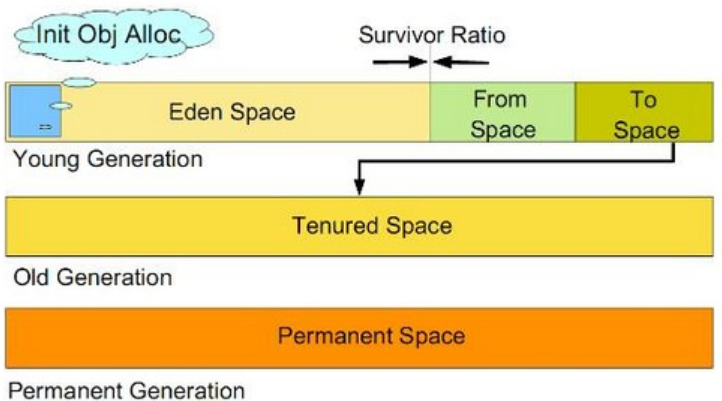
  明白这些我们基本明白JVM如何正确回收，接下来就是JVM什么情况下触发GC以及GC触发的方式?

  第一个问题比较容易回答当然是当内存空间不足的时候就需要触发GC，GC回收的时候采用的是分代收集的算法，主要分为年轻代和老年代，接下来我们简单介绍一下这2种方式：

年轻代：当一个对象被创建的时候，内存分配首先分配在年轻代，大部分对象创建以后都不再使用，对象很快变得不可达，就是对象无用，由于垃圾是被年轻代清理掉的，所以被叫做Minor GC或者Young GC。

   老年代:对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时，将执行Major GC，也叫 Full GC。

   明白这2块主要存放什么东西以后接下来我们看下GC的整体结构，看一个对象如何被Kill掉的流程：



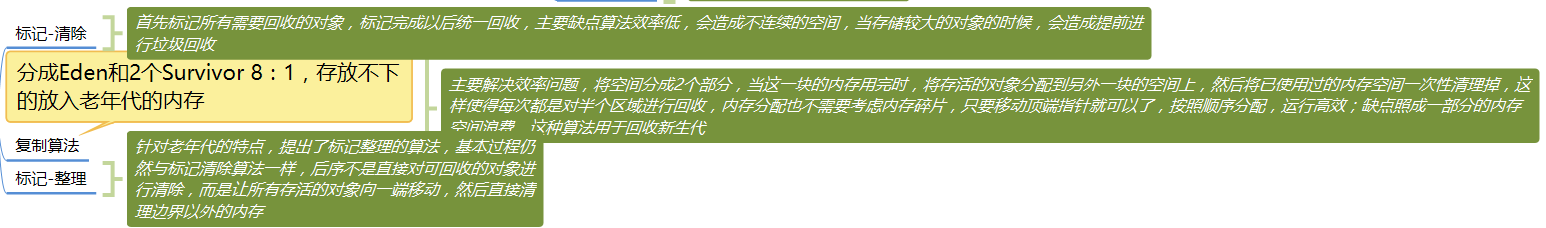
   1.当一个对象被创建的时候(new)首先会在年轻代的Eden区被创建，直到当GC的时候，根据可达性算法，看一个对象是否消亡，没有消亡的对象会被放入年轻带的Survivor（存活区）区，消亡的直接被Minor GC Kill掉；

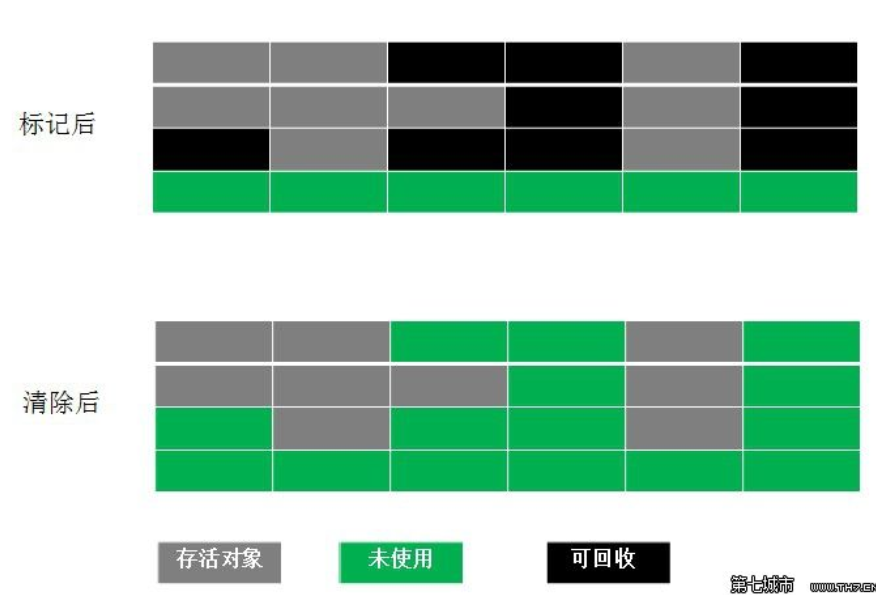
   2.进入到Survivor区的对象也不是安全的，当下一次Minor GC来的时候还是会检查Enden和Survivor存放对象区域中对象是否存活，存活放入另外一块Survivor区域；

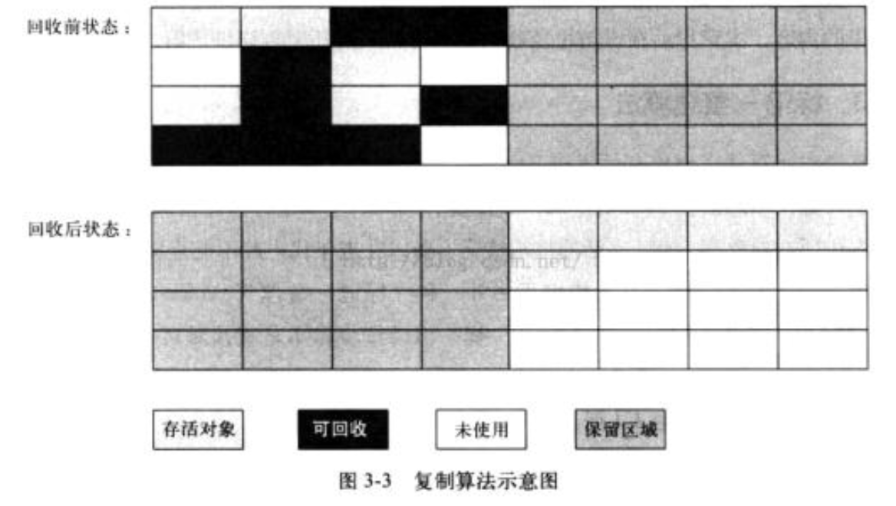
   3.当2个Survivor区切换几次以后，会直接进入老年代，当然进入到老年代也不是安全的，当老年代内存空间不足的时候，会触发Major GC,已经消亡的依然还是被Kill掉；

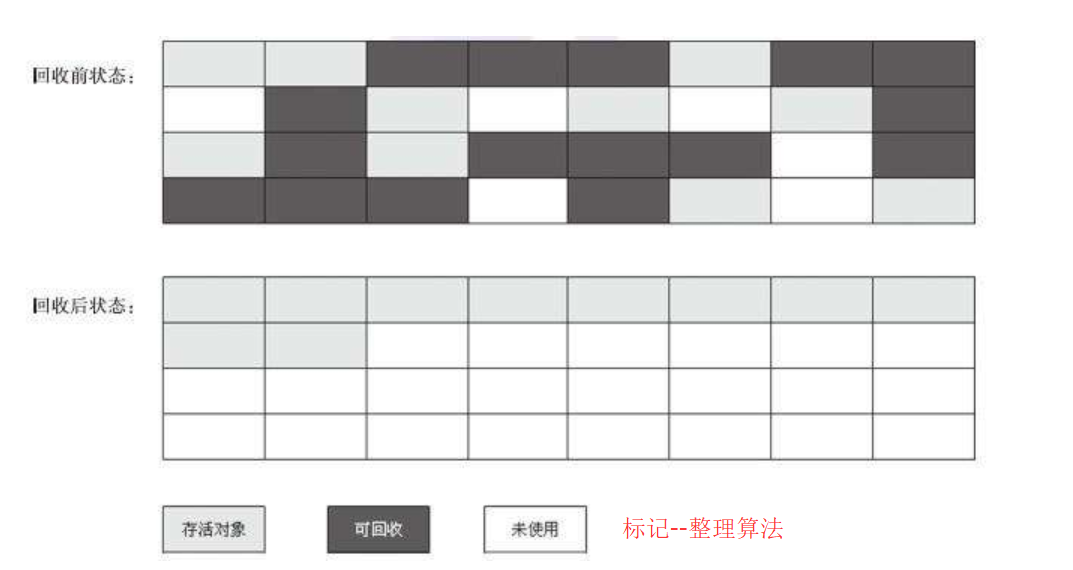
   推荐一个这个写的很逗可以看下：http://blog.csdn.net/sd4015700/article/details/50109939

   接下来我们还需要说一下GC的算法：标记--清除，复制，标记--整理这3种算法；









了解算法和GC内存分布以后我们接下来介绍垃圾回收器，这部分内容我不计划用文字去介绍，在第三个栏目我会将我对《深入理解Java虚拟机》这本书的思维导图，内容还不是很完善我正在整理中，但是有GC这部分内容包括各种参数配置，大家可以下载下来具体了解一下；

    最后我们谈一下监控和优化，当年具备以上知识以后这些都将不是问题，所以工欲善其事必先利其器，这就是我要说的，剩下就是对工具操作，这些我认为不需要介绍也是可以的，当然我也推荐一个博客：http://blog.csdn.net/renfufei/article/details/56678064

#### 问：内存泄露额内存移除

答：

概念：

内存溢出指的是内存不够用了。

内存泄漏是指对象可达，但是没用了。即本该被GC回收的对象并没有被回收

内存泄露是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。

内存泄漏的原因分析：

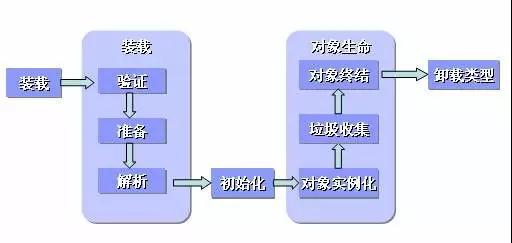
长生命周期的对象引用短生命周期的对象

没有将无用对象置为null

#### 问：简述一下Java中创建一个对象的过程？

1.检测类是否被加载—> 2.为对象分配内存 --> 3.为分配的内存空间初始化零值 -->4.对对象进行其他设置 --> 5 执行init方法

下图展示的是类的生命周期流向：



New关键字创建对象的过程



## 一、Spring

### 1.Java Bean，实现了Bean的规范的类就是javabean。

\*必须是个公共类（public）

\*有无参构造函数

\*用公共方法暴露内部成员属性(getter、setter)

#### Bean的作用域

Spring定义了多种Bean的作用域，可以基于这些作用域创建bean，包括

**\* 单利（Singleton）：**在整个应用中，只创建bean的一个实例。

**\* 原型（Prototype）：**每次注入或者通过Spring应用上下文获取的时候，都会创建一个新的bean实例。

**\* 回话（Session）：**在web应用中，为每个会话创建一个bean实例。

**\* 请求（Request）：**在web应用中，为每个请求创建一个bean实例。

在代码里可以如下:

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class MyIsBean{….}

XML版本：

<bean id=”beanName”

class=”xxx.xx.xxx.beans”>

scope=”prototype”

/>

在默认情况下，Spring应用上下文所有bean都是做为单利（Singleton）的形式创建的。也就是说，不管给定的一个bean被注入到其他的bean多少次，每次注入的都是同一个实例。

在大多数情况下，单例bean是很理想的方案。初始化和垃圾回收对象实例所带来的成本只留给一些小规模任务，在这些任务中，让对象保持无状态并且在应用中反复重用这些对象可能并不合理。

有时候，可能会发现，你所使用的类是易变的（mutable），它们会保持一些状态，因此重用是不安全的。在这种情况下，将class声明为单例的bean就不是什么好主意了，因为对象会被污染，稍后重用的时候会出现意想不到的问题。

#### 声明Bean

以下是声明bean的注解：

@Component组件，没有明确的角色

@Service在业务逻辑层使用

@Repository在数据访问层使用

@Controllor在展现层使用（MVC->Spring MVC）使用

\* 在这里，可以指定bean的id名：Component(“xxxx”)

\* 同时，Spring支持将@Named作为@Component注解的替代方案。两者之间有一些细微的差异，但是在大多数场景中是可以相互替换的。

### DI-依赖注入

#### 注入bean的注解

**@Autowired Spring提供的注解**

不仅仅是对象，还有构造器上，还能用在属性的Setter方法上。

不管是构造器、Setter方法还是其他的方法，Spring都会尝试满足方法参数上所声明的依赖。假如有且只有一个bean匹配依赖需求的话，那么这个bean将会被装配进来。

如果没有匹配的bean，那么在应用上下文创建的时候，Spring会抛出一个异常。为了避免异常的出现，你可以将@Autowired的required属性设置为false。

将required属性设置为false时，Spring会尝试执行自动装配，但是如果没有匹配的bean的话，Spring将会让这个bean处于未装配的状态。但是，把required属性设置为false时，你需要谨慎对待。如果在你的代码中没有进行null检查的话，这个处于未装配状态的属性有可能会出现NullPointerException。

@Inject注解来源于Java依赖注入规范，该规范同时还为我们定义了@Named注解。在自动装配中，Spring同时支持@Inject和@Autowired。尽管@Inject和@Autowired之间有着一些细微的差别，但是在大多数场景下，它们都是可以互相替换的。

@Autowired 是最常见的注解之一，但在老项目中，你可能会看到这些注解，它们的作用和@Autowired 相近：

* @Inject 是JSR-330提供的注解
* @Resource 是JSR-250提供的注解

#### 条件化的Bean

假设你希望一个或多个bean只有在应用的类路径下包含特定的库时才创建。或者我们希望某个bean只有当另外某个特定的bean也声明了之后才会创建。我们还可能要求只有某个特定的环境变量设置之后，才会创建某个bean。

在Spring 4之前，很难实现这种级别的条件化配置，但是Spring 4引入了一个新的@Conditional注解，它可以用到带有@Bean注解的方法上。如果给定的条件计算结果为true，就会创建这个bean，否则的话，这个bean会被忽略。

通过ConditionContext，我们可以做到如下几点：

* 借助getRegistry()返回的BeanDefinitionRegistry检查bean定义；
* 借助getBeanFactory()返回的ConfigurableListableBeanFactory检查bean是否存在，甚至探查bean的属性；
* 借助getEnvironment()返回的Environment检查环境变量是否存在以及它的值是什么；
* 读取并探查getResourceLoader()返回的ResourceLoader所加载的资源；
* 借助getClassLoader()返回的ClassLoader加载并检查类是否存在。

#### 处理自动装配的歧义性

#### 标示首选的bean

在声明bean的时候，通过将其中一个可选的bean设置为首选（primary）bean能够避免自动装配时的歧义性。当遇到歧义性的时候，Spring将会使用首选的bean，而不是其他可选的bean。实际上，你所声明就是“最喜欢”的bean。

#### 限定自动装配的bean

设置首选bean的局限性在于@Primary无法将可选方案的范围限定到唯一一个无歧义性的选项中。它只能标示一个优先的可选方案。当首选bean的数量超过一个时，我们并没有其他的方法进一步缩小可选范围。

与之相反，Spring的限定符能够在所有可选的bean上进行缩小范围的操作，最终能够达到只有一个bean满足所规定的限制条件。如果将所有的限定符都用上后依然存在歧义性，那么你可以继续使用更多的限定符来缩小选择范围。

@Qualifier注解是使用限定符的主要方式。它可以与@Autowired和@Inject协同使用，在注入的时候指定想要注入进去的是哪个bean。例如，我们想要确保要将IceCream注入到setDessert()之中：

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Autowired  @Qualifier("iceCream")  public void setDessert(Dessert dessert){    this.dessert = dessert;  } |

这是使用限定符的最简单的例子。为@Qualifier注解所设置的参数就是想要注入的bean的ID。所有使用@Component注解声明的类都会创建为bean，并且bean的ID为首字母变为小写的类名。因此，@Qualifier(“iceCream”)指向的是组件扫描时所创建的bean，并且这个bean是IceCream类的实例。

实际上，还有一点需要补充一下。更准确地讲，@Qualifier(“iceCream”)所引用的bean要具有String类型的“iceCream”作为限定符。如果没有指定其他的限定符的话，所有的bean都会给定一个默认的限定符，这个限定符与bean的ID相同。因此，框架会将具有“iceCream”限定符的bean注入到setDessert()方法中。这恰巧就是ID为iceCream的bean，它是IceCream类在组件扫描的时候创建的。

基于默认的bean ID作为限定符是非常简单的，但这有可能会引入一些问题。如果你重构了IceCream类，将其重命名为Gelato的话，那此时会发生什么情况呢？如果这样的话，bean的ID和默认的限定符会变为gelato，这就无法匹配setDessert()方法中的限定符。自动装配会失败。

这里的问题在于setDessert()方法上所指定的限定符与要注入的bean的名称是紧耦合的。对类名称的任意改动都会导致限定符失效。

#### SpringEL

* Value实现资源的注入

#### Bean的初始化和销毁

* Java配置方式：initMethod和destoryMethod
* 注解：@PostConstruct和@PreDestory

#### Profile

提供在不同的环境下使用不同的配置

##### 激活Profile

Spring在确定哪个profile处于激活状态时，需要依赖两个独立的属性：spring.profiles.active和spring.profiles.default。如果设置了spring.profiles.active属性的话，那么它的值就会用来确定哪个profile是激活的。但如果没有设置spring.profiles.active属性的话，那Spring将会查找spring.profiles.default的值。如果spring.profiles.active和spring.profiles.default均没有设置的话，那就没有激活的profile，因此只会创建那些没有定义在profile中的bean。

##### 使用profile进行测试

当运行集成测试时，通常会希望采用与生产环境（或者是生产环境的部分子集）相同的配置进行测试。但是，如果配置中的bean定义在了profile中，那么在运行测试时，我们就需要有一种方式来启用合适的profile。

Spring提供了@ActiveProfiles注解，我们可以使用它来指定运行测试时要激活哪个profile。在集成测试时，通常想要激活的是开发环境的profile。

比如Profile(“dev”)

#### Application Event

使用Application Event可以做到Bean与Bean之间的通信

Spring的事件需要遵循如下流程：

* 自定义事件，集成ApplicationEvent
* 定义事件监听器，实现ApplicationListener
* 使用容器发布事件

### AOP-面向切面编程

#### @AspectJ

#### 切点标志符(designator)

AspectJ5 的切点表达式由标志符(designator)和操作参数组成. 如 "execution(greetTo(..))" 的切点表达式,***execution***就是 标志符, 而圆括号里的greetTo(..) 就是操作参数

##### execution

匹配 join point 的执行, 例如 "execution(\* hello(..))" 表示匹配所有目标类中的 hello() 方法. 这个是最基本的 pointcut 标志符.

##### within

匹配特定包下的所有 join point, 例如 within(com.xys.\*) 表示 com.xys 包中的所有连接点, 即包中的所有类的所有方法. 而within(com.xys.service.\*Service) 表示在 com.xys.service 包中所有以 Service 结尾的类的所有的连接点.

##### this 与 target

this 的作用是匹配一个 bean, 这个 bean(Spring AOP proxy) 是一个给定类型的实例(instance of). 而 target 匹配的是一个目标对象(target object, 即需要织入 advice 的原始的类), 此对象是一个给定类型的实例(instance of).

##### bean

匹配 bean 名字为指定值的 bean 下的所有方法, 例如:

bean(\*Service) // 匹配名字后缀为 Service 的 bean 下的所有方法

bean(myService) // 匹配名字为 myService 的 bean 下的所有方法

##### args

匹配参数满足要求的的方法.

package com.hqj.aop;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

/\*\*

\* 横切关注点 point

\* @author Administrator

\*

\*/

@Aspect

@Component

public class TimeHandler {

/\*\*

\* 一：

\* execution 匹配 join point 的执行, 例如 "execution(\* hello(..))" 表示匹配所有目标类中的 hello() 方法.

\* 这个是最基本的 pointcut 标志符.

\*/

// @Before("execution(\* com.hqj.service\*..\*User\*(..))")

// @Before("execution(\* com.hqj.service\*..\*User\*.delete\*(..))")

/\*\*

\* 二：

\* 匹配特定包下的所有 join point, 例如 within(com.xys.\*) 表示 com.xys 包中的所有连接点, 即包中的所有类的所有方法.

\* 而 within(com.xys.service.\*Service) 表示在 com.xys.service 包中所有以 Service 结尾的类的所有的连接点.

\*/

// @Before("within(com.hqj.service.\*)")

/\*\*

\* 三：

\* bean

\* 匹配 bean 名字为指定值的 bean 下的所有方法, 例如:

\* bean(\*Service) // 匹配名字后缀为 Service 的 bean 下的所有方法

\* bean(myService) // 匹配名字为 myService 的 bean 下的所有方法

\*/

// @Before("bean(UserService)")

/\*\*

\* 四：

\* this 的作用是匹配一个 bean, 这个 bean(Spring AOP proxy) 是一个给定类型的实例(instance of).

\* 而 target 匹配的是一个目标对象(target object, 即需要织入 advice 的原始的类), 此对象是一个给定类型的实例(instance of).

\*/

// @Before("execution(\* com.hqj.service.User\*..\*(..)) && this(com.hqj.service.UserService)")

// @Before("target(com.hqj.service.UserService)")

// public void before(){

// System.out.println("AOP @Before Time : " + System.currentTimeMillis());

// }

/\*\*

\* 五：

\* args 匹配参数满足要求的的方法.

\* 切点方法需要与织入的接口方法参数一致

\* 参数织入能知道你在操作的数据参数值

\*/

// @Before("execution(\* com.hqj.service\*..\*User\*.get\*(..)) && args(id,name)")

// public void before(Integer id,String name){

// System.out.println("AOP @Before Time : " + System.currentTimeMillis() + " id :" + id + " name :" + name);

// }

/\*\*

\* 六：

\* annotation 匹配由指定注解所标注的方法.

\*/

@Before("annotation(myTag)")

public void before(){

System.out.println("AOP @Before Time : " + System.currentTimeMillis());

}

@After("execution(\* com.hqj.service\*..\*User\*(..))")

public void after(){

System.out.println("AOP @After Time : " + System.currentTimeMillis());

}

}

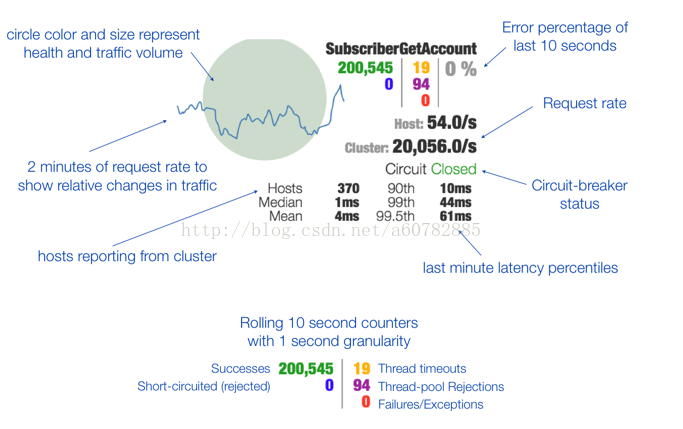
## 二、SpringMVC

## 三、Spring boot

## 四、spring cloud

### 1：加入Turbine熔断监控的必要性

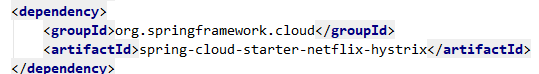
微服务间通过http协议获取服务定然会有网络因素、服务启停引起的服务故障概率。但服务间的调用情况对我们而言就好像是蒙住眼睛闻菜香，味道和实际可能并不一致。Turbine可以展示服务间调用的详细情况以及统计、线程池的使用情况：



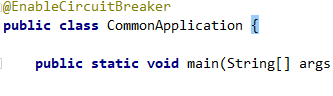


**集成方式：**

1. maven引入hystrix依赖：

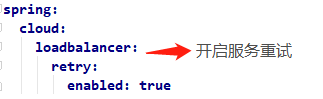


1. 开启断路器

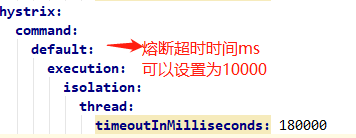


1. 确保服务间调用开启了hystrix：

application.yml中加入配置：

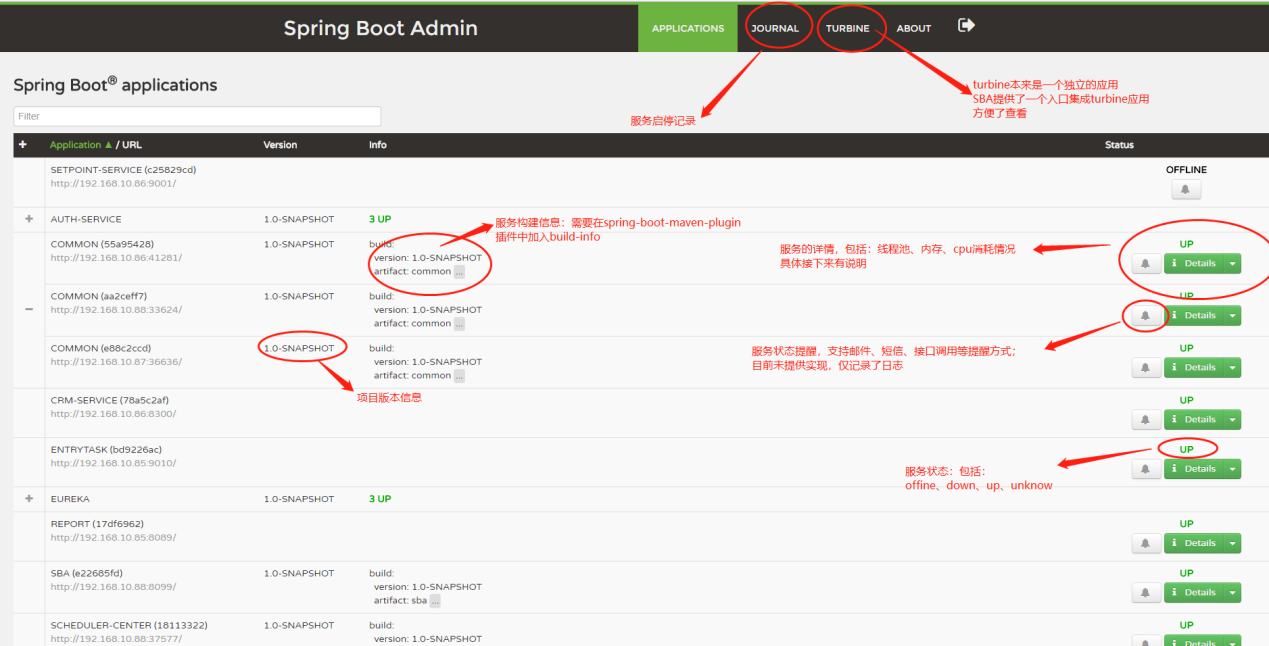








### 2：整合Spring Boot Admin为什么要集成SBA；



1.由于需要访问服务的JVM、内存、CPU、线程池等使用情况，需要验证，由于内部服务都在内网段内，去掉安全验证降低集成复杂度：



2.客户端展示信息完善：展示版本号等项目构建信息



1. 网关静态资源路由路径规范,

对于entrytask、crm、auth需要由zuul路由静态资源的项目，需要按如下路径配置aliyun oss的资源路径：

以crm项目为例：

开发环境：blm/dsp/crm/web/dev

测试环境：blm/dsp/crm/web/test

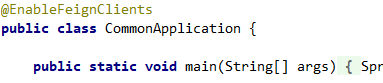
预发布环境：blm/dsp/crm/web/beta

生产环境：blm/dsp/crm/web/release

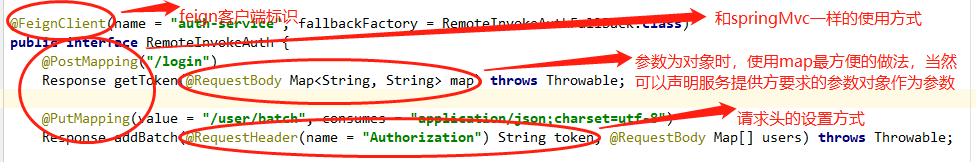
### 3：服务间feign调用语法以及简单示例

1. 开启feign

启动类加入注解

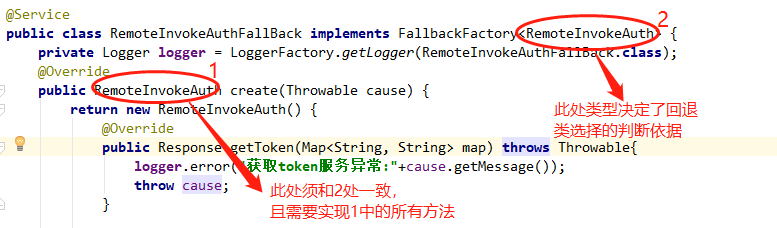


1. feign接口标识



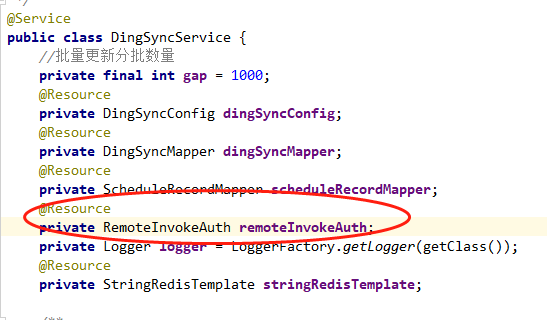
@FeignClient标识此接口为微服务调用接口，参数name位服务提供者的服务名，fallbackFactory为服务调用失败后回退的类，一般是为了更好的用户体验和自定义回退逻辑

1. 回退类



1. 调用服务方式

将第2步声明的RemoteInvokeAuth作为普通Bean注入到需要调用服务的类，直接使用即可。





## 五、Velocity

## 六、Mybitis

## 七、Hibernate

## 八、WebService

## 九、XML

## 十、JSON

## 十一、Oracle

## 十二、SQL Server

## 十二·五、各个数据源比较

## 十三、MySql

### （一）数据库三大范式

第一范式，**1NF是对属性的原子性约束，要求属性具有原子性，不可再分解**；通俗的理解是，字段还可以再分吗？如果不能，则符合1NF的设计。

第二范式，**2NF是对记录的唯一性约束，要求记录有唯一标识，即实体的唯一性**；简单的解释就，比如你和一个女生约会会建立一张表，不用每条记录都会记录她的身高、体重，将身高、体重单独的存在一张表中供查询即可。

第三范式， **3NF是对字段的冗余性的约束，即任何字段不能由其他字段派生出来，它要求没有冗余。**打个比方，比如评论表，如果你将用户ID，用户头像都放在这留言表中，就是不合适了。用户头像是依赖于用户ID，而不依赖该评论。

### （二）经典SQL数据库

#### 基础SQL

创建数据库 Create Databas NAME

删除数据库 Drop DataBase NAME

创建表

Create Table TableName（

Id int（10），

列名 类型（大小）

）；

删除表

Drop Table TableName

#### 常用的SQL

查询：Select \*或者（列名1，列名2…） from Table Where 条件（范围）

插入：Inset into table（列名1，列名2,….） values（值1，值2….）

更新：Update Table Set 列名1=值1 ，列名2=值2…. Where 条件（范围）

查找：Select \*或者（列名1，列名2…） from Table Where 列名 **LIKE** “ %value1%” ….；

排序：Select \*或者（列名1，列名2…） from Table Where 条件 **ORDER BY** ID,DATA **DESC**;

总数：select **Count**（） from Table

**Count（）可以和 DISTINCT ：Select Count（DISTINCT 列名） FROM TABLE**



求和：Select SUM（列名） FROM Table

平均：Select AVG（列名） FROM TABLE

最大：select MAX（列名） FROM TABLE

最小：Select MIN（列名） FROM TABLE

#### 几个高级的查询运算词

##### UNION、UNION ALL操作符

用于合并两个或多个 SELECT 语句的结果集。

请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

**语法：**

Select \* From TableA **UNION** Select \* FROM TableB



##### EXCEPT操作符

EXCEPT 返回两个结果集的差（即从左查询中返回右查询没有找到的所有非重复值）。

##### INTERSECT操作符

返回 两个结果集的交集（即两个查询都返回的所有非重复值）

##### 使用外连接（左left 右right连接）

Select a.a,a.c..,b.d,d.e… FROM a LEFT JOIN b ON a.a = b.d（返回左表的集合）

Select a.a,a.c..,b.d,d.e… FROM a RIGHT JOIN b ON a.a = b.d（返回右表的集合）

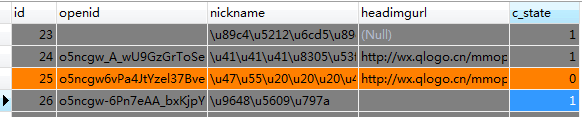
##### 分组:Group by

一张表，一旦分组 完成后，查询后只能得到组相关的信息。

**组相关的信息：（统计信息） count,sum,max,min,avg 分组的标准)**

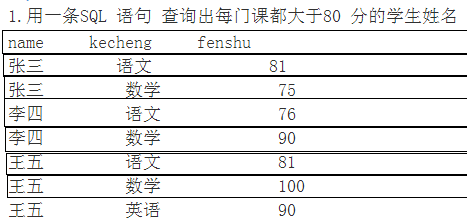
在SQLServer中分组时：不能以text,ntext,image类型的字段作为分组依据

在selecte统计函数中的字段，不能和普通的字段放在一起；



select count(id) FROM t\_commenter WHERE c\_state = 1 **Group by** c\_state

##### SQL经典例子



SELECT name1,fenshu from table1 group by name1 having min(fenshu)>80

思路，按name分组 使用having包换 min()最小值 大于80的

### （三）数据库锁

[**谈谈mysql的悲观和乐观锁**](https://www.cnblogs.com/phpper/p/7345332.html)

悲观锁与乐观锁是两种常见的资源并发锁设计思路，也是并发编程中一个非常基础的概念。之前有写过一篇文章[关于并发的处理思路](http://www.cnblogs.com/phpper/p/6716248.html)和解决方案，这里我单独将对这两种常见的锁机制在数据库数据上的实现进行比较系统的介绍一次吧。

#### 悲观锁（Pessimistic Lock）

悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常所说的“一锁二查三更新”即指的是使用悲观锁。通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的select … for update操作来实现悲观锁。当数据库执行select for update时会获取被select中的数据行的行锁，因此其他并发执行的select for update如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

这里需要注意的一点是不同的数据库对select for update的实现和支持都是有所区别的，例如oracle支持select for update no wait，表示如果拿不到锁立刻报错，而不是等待，mysql就没有no wait这个选项。另外mysql还有个问题是select for update语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此如果在mysql中用悲观锁务必要确定走了索引，而不是全表扫描。

#### 乐观锁（Optimistic Lock）

乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号，或者时间戳，然后按照如下方式实现：

[复制代码](javascript:void(0);)

1. SELECT data AS old\_data, version AS old\_version FROM …;

2. 根据获取的数据进行业务操作，得到new\_data和new\_version

3. UPDATE SET data = new\_data, version = new\_version WHERE version = old\_version

if (updated row > 0) {

// 乐观锁获取成功，操作完成

} else {

// 乐观锁获取失败，回滚并重试

}

[复制代码](javascript:void(0);)

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句时会获取被update行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这之间没有发生并发的修改。如果更新失败即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

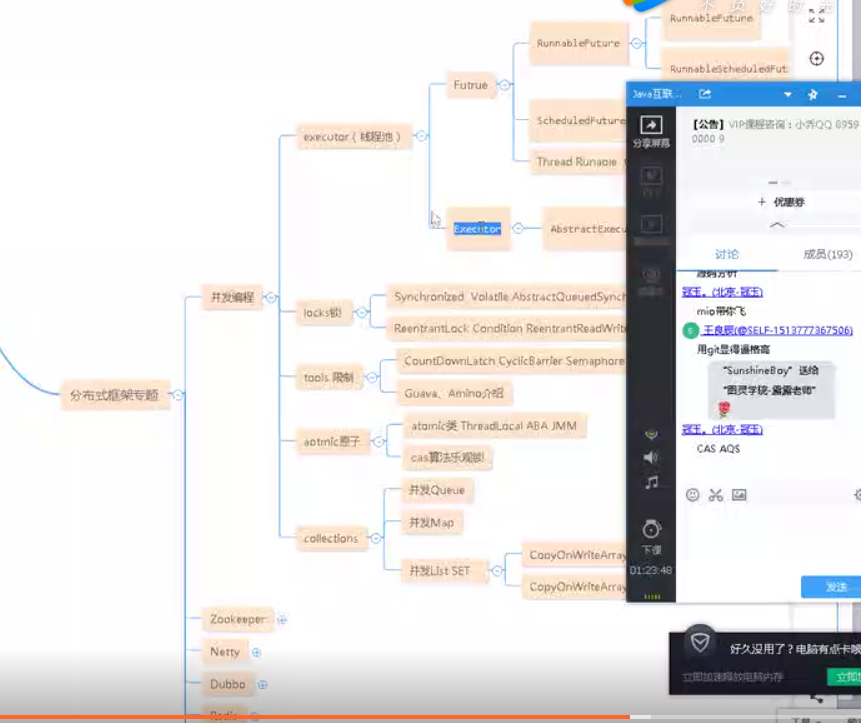
#### 总结

* 乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能
* 乐观锁还适用于一些比较特殊的场景，例如在业务操作过程中无法和数据库保持连接等悲观锁无法适用的地方

### （四）数据库事务

## 十四、PostgreSQL

## 十五、分布式系统的设计、分布式、缓存、消息、负载均衡等机制的实现



## 十六、Linux开发环境

## 十七、大数据处理及分布式管理前沿技术（Hadoop、MapReduce、Spark、ZooKeeper、NoSQL框架）

### 1.Zookeeper

### 2.NoSql（Redis、MongoDB、HBase[大数据用]）

### 3.Hadoop

### 4.MapReduce

### 5.Spark

## 十八、Nginx服务器

## 十九、Maven、SVN、GIt

### （一）、Maven

### ****第1章 课程概述****

* [1-1 项目管理利器-maven课程概述 (02:51)](http://www.imooc.com/video/8597)

### ****第2章 唯快不破-maven快速入门****

* [2-1 maven介绍及环境搭建 (05:01)](http://www.imooc.com/video/8598)
* [2-2 第一个maven案例hellomaven (13:22)](http://www.imooc.com/video/8599)

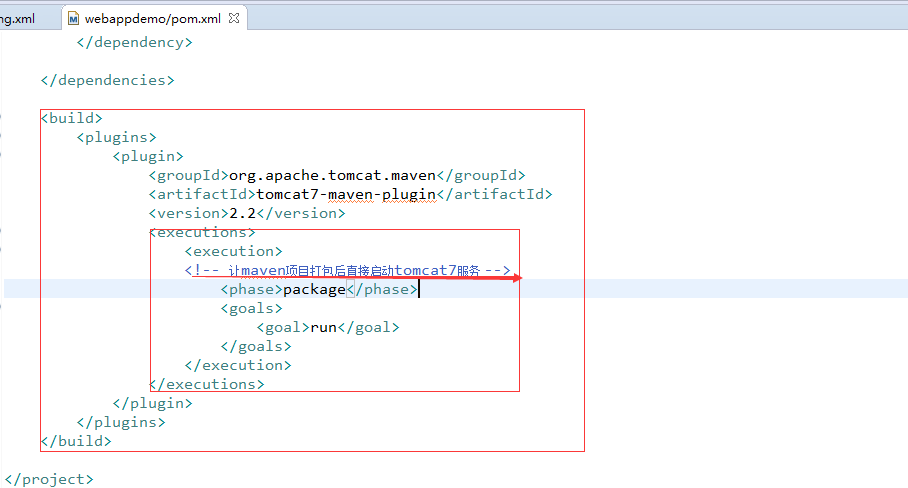
Eclipse搭建、创建Maven项目

### ****第3章 稳扎稳打--Maven核心知识****

* [3-1 maven常用的构建命令 (12:10)](http://www.imooc.com/video/8615)

compile（编译） clean（删除target目录下编译内容） install（在本地的repository安装jar） package（打war包） test

让maven项目打包后直接启动服务的方式如图



* [3-2 maven自动建立目录骨架 (07:31)](http://www.imooc.com/video/8601)
* [3-3 maven中的坐标和仓库 (12:01)](http://www.imooc.com/video/8652)

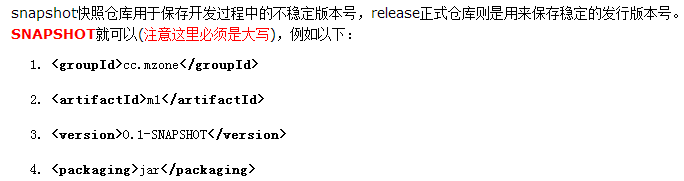
坐标

由<groupId>（一般为域名倒写+公司名称+项目名称）</groupId>

<artifactId>（项目名称）</artifactId>

<version>（版本号SNAPSHOT）</version>

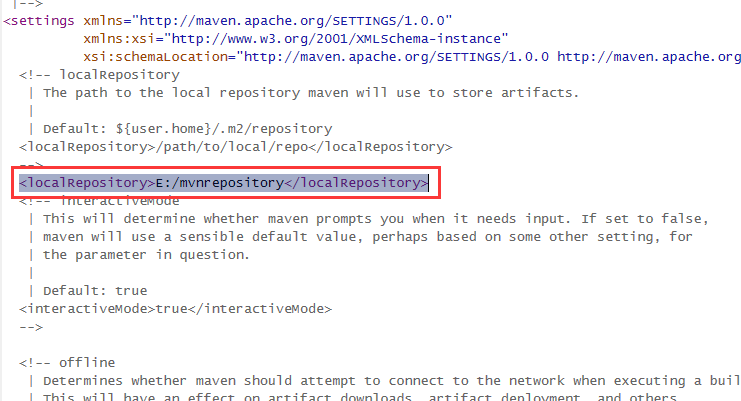
三个确定坐标



构件

仓库

本地仓库（maven配置文件settings.xml修改）



中央仓库：<https://repo.maven.apache.org/maven2>

镜像仓库：http://www.mvnrepository.com/

更改仓库位置：

* [3-4 在eclipse中安装maven插件以及创建maven项目 (08:20)](http://www.imooc.com/video/8643)
* [3-5 maven的生命周期和插件 (08:03)](http://www.imooc.com/video/8644)
* [3-6 maven中Pom.xml解析 (12:34)](http://www.imooc.com/video/8645)
* [3-7 maven依赖范围 (04:18)](http://www.imooc.com/video/8648)
* [3-8 maven依赖传递 (09:24)](http://www.imooc.com/video/8649)
* [3-9 maven依赖冲突 (04:13)](http://www.imooc.com/video/8650)
* [3-10 maven聚合和继承 (06:48)](http://www.imooc.com/video/8651)

### ****第4章 如何构建web项目****

* [4-1 使用maven构建web项目 (11:14)](http://www.imooc.com/video/8684)

### ****第5章 课程总结****

* 最近学习[5-1 课程总结 (02:47)](http://www.imooc.com/video/8613)

## 二十、Redis、Memcache（面试角度）

### （1）、Redis

#### 1.什么是redis？

Redis是一款C语言编写的，基于key-valu存储的内存数据库，支持String、list、set、zset（sorted set）、hash。

#### 2.redis的特点

**优点：性能，数据结构，异步存盘。**

**缺点：受限于内存，不能进行海量数据的高性能读写。**

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。  
Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种[数据结构](http://lib.csdn.net/base/datastructure)，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一个功能加强版的memcached来用。  
Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

#### 3.Redis支持的数据类型

Redis通过Key-Value的单值不同类型来区分，以下是支持的类型：

String

List

Set

Sorted Set ( Zset)

hash

#### 4.Redis是单进程单线程的

redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

## 二十一、IO、NIO、RPC

### 1. IO

所谓IO，也就是Input与Output的缩写。在java中，IO涉及的范围比较大，这里主要讨论针对文件内容的读写。

对文件内容的操作主要分两大类

分别是：

**字符流**

**字节流**

其中，**字符流**有两个抽象类：**Writer Reader**

对应子类**FileWriter**和**FileReader**可实现文件的读写操作

**BufferWriter**和**BufferReader** 能够提供缓冲区功能，可以提高效率

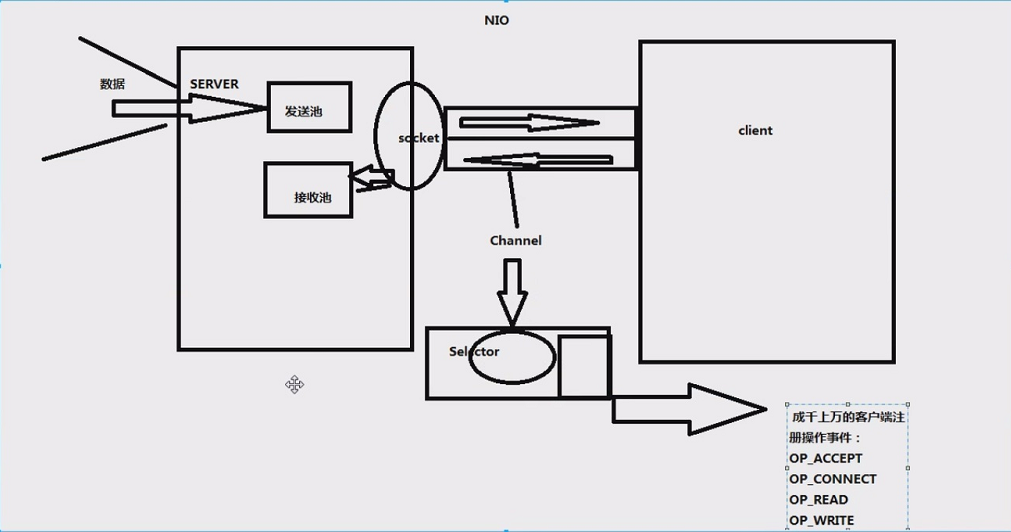
同样，**字节流**也有两个抽象类：**InputStream OutputStream**

其对应的子类有**FileInputStream**和**FileOutputStream**实现文件读写

**BufferedInputStream** 和 **BufferedOutputStream** 提供缓冲区功能

### 2.NIO

#### （一） Java NIO 概述



Java NIO 由以下几个核心部分组成：

**Channels**

**Buffers**

**Selectors**

虽然Java NIO 中除此之外还有很多类和组件，但在我看来，Channel，Buffer 和 Selector 构成了核心的API。其它组件，如Pipe和FileLock，只不过是与三个核心组件共同使用的工具类。因此，在概述中我将集中在这三个组件上。其它组件会在单独的章节中讲到。

##### Channel 和 Buffer

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。这里有个图示：

****

Channel和Buffer有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：

**FileChannel（文件IO）**

**DatagramChannel（UDP）**

**SocketChannel（TCP）**

**ServerSocketChannel（TCP服务端）**

正如你所看到的，这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。

与这些类一起的有一些有趣的接口，但为简单起见，我尽量在概述中不提到它们。本教程其它章节与它们相关的地方我会进行解释。

以下是Java NIO里关键的Buffer实现：

**ByteBuffer**

**CharBuffer**

**DoubleBuffer**

**FloatBuffer**

**IntBuffer**

**LongBuffer**

**ShortBuffer**

这些Buffer覆盖了你能通过IO发送的基本数据类型：byte, short, int, long, float, double 和 char。

Java NIO 还有个 MappedByteBuffer，用于表示内存映射文件， 我也不打算在概述中说明。

##### Selector

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：



要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

#### （二） Channel

Java NIO的通道类似流，但又有些不同：

既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。

通道可以异步地读写。

通道中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。

正如上面所说，从通道读取数据到缓冲区，从缓冲区写入数据到通道。如下图所示：



##### Channel的实现

这些是Java NIO中最重要的通道的实现：

**FileChannel**

**DatagramChannel**

**SocketChannel**

**ServerSocketChannel**

FileChannel 从文件中读写数据。

DatagramChannel 能通过UDP读写网络中的数据。

SocketChannel 能通过TCP读写网络中的数据。

ServerSocketChannel可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel。

##### 基本的 Channel 示例

下面是一个使用FileChannel读取数据到Buffer中的示例：

/\*\*

\* ByteBuffer

\* clear()使缓冲区准备好信道读取或相对放置操作的一个新的序列：它设置了限制的能力和位置为零。

\* flip()使缓冲区准备好新的通道写入或相对获取操作序列：它将限制设置为当前位置，然后将位置设置为零。

\* rewind()使缓冲区准备好重新读取已经包含的数据：它保持限制不变，并将位置设置为零。

\*/

//使用RandomAccessFile 读取磁盘文件

RandomAccessFile aFile = **new** RandomAccessFile("G:\\a.txt", "rw");

//获取文件FileChannel对象 FileChannel

FileChannel inChannel = aFile.getChannel();

//申请一个ByteBuffer空间

ByteBuffer buf = ByteBuffer.*allocate*(48);

**int** bytesRead = inChannel.read(buf);

**while** (bytesRead != -1) {

System.***out***.println("Read " + bytesRead + "\n");

buf.flip();//切换读模式

//buf.hasRemaining() true如果，此缓冲区中至少有一个元素

**while** (buf.hasRemaining()) {

System.***out***.print((**char**) buf.get());

}

buf.clear();

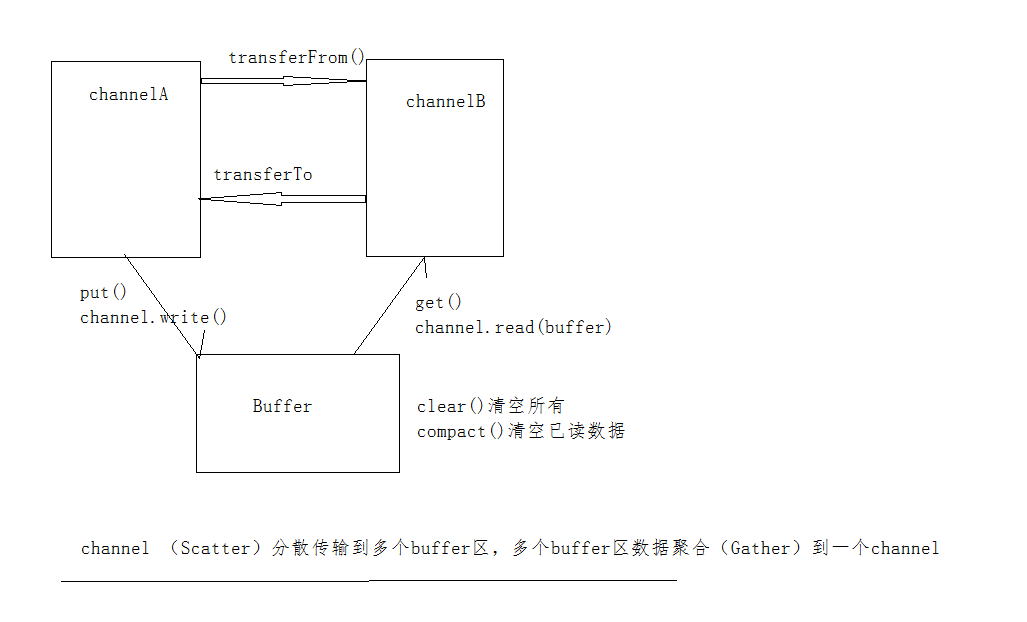
bytesRead = inChannel.read(buf);

}

aFile.close();

注意 buf.flip() 的调用，首先读取数据到Buffer，然后反转Buffer,接着再从Buffer中读取数据。下一节会深入讲解Buffer的更多细节。

#### （三） Buffer flip()、get()、put()、clear()、compact()



**当向buffer写入数据时，buffer会记录下写了多少数据。一旦要读取数据，需要通过flip()方法将Buffer从写模式切换到读模式。在读模式下，可以读取之前写入到buffer的所有数据。一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用clear()或compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。**

flip() 将buffer的写模式切换到读模式。

get() 获取buffer中的数据

put()

clear() 将清空整个buffer缓冲区的数据。

compact() 清空已读数据，未读数据会被放到缓冲区起始区，任何新加入的数据会被加到未读数据后。

Java NIO中的Buffer用于和NIO通道进行交互。如你所知，数据是从通道读入缓冲区，从缓冲区写入到通道中的。

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。

下面是NIO Buffer相关的话题列表：

1. [Buffer的基本用法](http://ifeve.com/buffers/#basicusage)
2. [Buffer的capacity,position和limit](http://ifeve.com/buffers/#cap-pos-limit)
3. [Buffer的类型](http://ifeve.com/buffers/#buffertype)
4. [Buffer的分配](http://ifeve.com/buffers/#allocate)
5. [向Buffer中写数据](http://ifeve.com/buffers/#writetobuffer)
6. [flip()方法](http://ifeve.com/buffers/#flip)
7. [从Buffer中读取数据](http://ifeve.com/buffers/#read)
8. [clear()与compact()方法](http://ifeve.com/buffers/#clearandcompact)
9. [mark()与reset()方法](http://ifeve.com/buffers/#markandreset)
10. [equals()与compareTo()方法](http://ifeve.com/buffers/#equalsandcompareto)

##### Buffer的capacity,position和limit

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。

为了理解Buffer的工作原理，需要熟悉它的三个属性：

**capacity （英 [kəˈpæsəti] 美 [kəˈpæsɪti] 容量）**

**position （英 [pəˈzɪʃn] 美 [pəˈzɪʃən] 坐标，位置;状态）**

limit

position和limit的含义取决于Buffer处在读模式还是写模式。不管Buffer处在什么模式，capacity的含义总是一样的。

这里有一个关于capacity，position和limit在读写模式中的说明，详细的解释在插图后面。



###### capacity

作为一个内存块，Buffer有一个固定的大小值，也叫“capacity”.你只能往里写capacity个byte、long，char等类型。一旦Buffer满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

###### Position

**position现在用于标记读的位置**

当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte、long等数据写到Buffer后， position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1.

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

###### Limit

**limit表示之前写进了多少个byte、char等 —— 现在能读取多少个byte、char等**

在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于Buffer的capacity。

当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读模式时，limit会被设置成写模式下的position值。换句话说，你能读到之前写入的所有数据（limit被设置成已写数据的数量，这个值在写模式下就是position）

##### Buffer的类型

Java NIO 有以下Buffer类型

**ByteBuffer**

**MappedByteBuffer**

**CharBuffer**

**DoubleBuffer**

**FloatBuffer**

**IntBuffer**

**LongBuffer**

**ShortBuffer**

如你所见，这些Buffer类型代表了不同的数据类型。换句话说，就是可以通过char，short，int，long，float 或 double类型来操作缓冲区中的字节。

MappedByteBuffer 有些特别，在涉及它的专门章节中再讲。

##### Buffer的分配

要想获得一个Buffer对象首先要进行分配。 每一个Buffer类都有一个**allocate**方法。下面是一个分配48字节capacity的ByteBuffer的例子。

|  |  |
| --- | --- |
|  | ByteBuffer buf = ByteBuffer.allocate(48); |

这是分配一个可存储1024个字符的CharBuffer：

|  |  |
| --- | --- |
|  | CharBuffer buf = CharBuffer.allocate(1024); |

##### 向Buffer中写数据

写数据到Buffer有两种方式：

从Channel写到Buffer。

通过Buffer的**put()**方法写到Buffer里。

从Channel写到Buffer的例子

|  |  |
| --- | --- |
| 1 | int bytesRead = inChannel.read(buf); //read into buffer. |

通过put方法写Buffer的例子：

|  |  |
| --- | --- |
| 1 | buf.put(127); |

put方法有很多版本，允许你以不同的方式把数据写入到Buffer中。

例如， 写到一个指定的位置，或者把一个字节数组写入到Buffer。 更多Buffer实现的细节参考JavaDoc。

###### flip()方法

flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。

换句话说，**position现在用于标记读的位置，limit表示之前写进了多少个byte、char等 —— 现在能读取多少个byte、char等**。

##### 从Buffer中读取数据

**从Buffer中读取数据有两种方式：**

1.从Buffer读取数据到Channel。

**Int byteWritten = inChannel.write(buf);**

2. 使用get()方法从Buffer中读取数据。

**byte b = buf.get();**

get方法有很多版本，允许你以不同的方式从Buffer中读取数据。例如，从指定position读取，或者从Buffer中读取数据到字节数组。更多Buffer实现的细节参考JavaDoc。

###### rewind()方法

Buffer.rewind()将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变，仍然表示能从Buffer中读取多少个元素（byte、char等）。

###### clear()与compact()方法

一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或compact()方法来完成。

如果调用的是clear()方法，position将被设回0，limit被设置成 capacity的值。换句话说，Buffer 被清空了。Buffer中的数据并未清除，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。

如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用compact()方法。

compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后面。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未读的数据。

##### mark()与reset()方法

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定position。之后可以通过调用Buffer.reset()方法恢复到这个position。

例如：

Buffer.mark()

Buffer.get();

Buffer.reset();

##### equals()与compareTo()方法

可以使用equals()和compareTo()方法两个Buffer。

###### equals()

当满足下列条件时，表示两个Buffer相等：

有相同的类型（byte、char、int等）。

Buffer中剩余的byte、char等的个数相等。

Buffer中所有剩余的byte、char等都相同。

如你所见，equals只是比较Buffer的一部分，不是每一个在它里面的元素都比较。实际上，它只比较Buffer中的剩余元素。

###### compareTo()方法

compareTo()方法比较两个Buffer的剩余元素(byte、char等)， 如果满足下列条件，则认为一个Buffer“小于”另一个Buffer：

第一个不相等的元素小于另一个Buffer中对应的元素 。

所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

*（译注：剩余元素是从 position到limit之间的元素）*

#### （四） Scatter/Gather

Java NIO开始支持scatter/gather，scatter/gather用于描述从Channel（译者注：Channel在中文经常翻译为通道）中读取或者写入到Channel的操作。

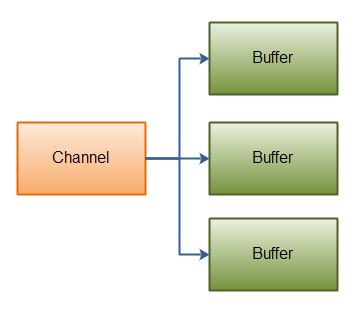
**分散（scatter）从Channel中读取是指在读操作时将读取的数据写入多个buffer中。因此，Channel将从Channel中读取的数据“分散（scatter）”到多个Buffer中。**

**聚集（gather）写入Channel是指在写操作时将多个buffer的数据写入同一个Channel，因此，Channel 将多个Buffer中的数据“聚集（gather）”后发送到Channel。**

scatter / gather经常用于需要将传输的数据分开处理的场合，例如传输一个由消息头和消息体组成的消息，你可能会将消息体和消息头分散到不同的buffer中，这样你可以方便的处理消息头和消息体。

**Scattering Reads**

Scattering Reads是指数据从一个channel读取到多个buffer中。如下图描述：

[](http://ifeve.com/java-nio-scattergather/scatter/)

Java NIO: Scattering Read

代码示例如下：

ByteBuffer bufferHead = ByteBuffer.*allocate*(128);

ByteBuffer bufferBody = ByteBuffer.*allocate*(1024);

ByteBuffer[] bufferArray = { bufferHead , bufferBody };

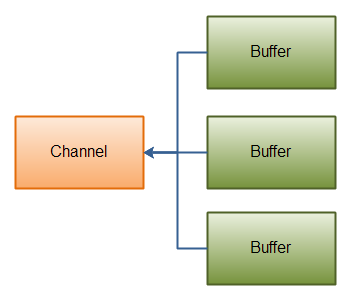
inChannel.read(bufferArray);

注意buffer首先被插入到数组，然后再将数组作为channel.read() 的输入参数。read()方法按照buffer在数组中的顺序将从channel中读取的数据写入到buffer，当一个buffer被写满后，channel紧接着向另一个buffer中写。

Scattering Reads在移动下一个buffer前，必须填满当前的buffer，这也意味着它不适用于动态消息(译者注：消息大小不固定)。换句话说，如果存在消息头和消息体，消息头必须完成填充（例如 128byte），Scattering Reads才能正常工作。

**Gathering Writes**

Gathering Writes是指数据从多个buffer写入到同一个channel。如下图描述：

[](http://ifeve.com/java-nio-scattergather/gather/)

Java NIO: Gathering Write

代码示例如下：

ByteBuffer bufferHead = ByteBuffer.*allocate*(128);

ByteBuffer bufferBody = ByteBuffer.*allocate*(1024);

ByteBuffer[] bufferArray = { bufferHead , bufferBody };

inChannel.write(bufferArray);

bufferArray数组是write()方法的入参，write()方法会按照buffer在数组中的顺序，将数据写入到channel，注意只有position和limit之间的数据才会被写入。因此，如果一个buffer的容量为128byte，但是仅仅包含58byte的数据，那么这58byte的数据将被写入到channel中。因此与Scattering Reads相反，Gathering Writes能较好的处理动态消息。

#### （五） 通道之间的数据传输

在Java NIO中，如果两个通道中有一个是FileChannel，那你可以直接将数据从一个channel（译者注：channel中文常译作通道）传输到另外一个channel。

##### transferFrom()

**FileChannel的transferFrom()方法可以将数据从源通道传输到FileChannel中**（译者注：这个方法在JDK文档中的解释为将字节从给定的可读取字节通道传输到此通道的文件中）。下面是一个简单的例子：

RandomAccessFile fromFile = **new** RandomAccessFile("G:\\a.txt", "rw");

RandomAccessFile toFile = **new** RandomAccessFile("G:\\b.txt","rw");

FileChannel formChannel = fromFile.getChannel();

FileChannel toChannel = toFile.getChannel();

**long** position = 0;

**long** count = formChannel.size();

toChannel.transferFrom(formChannel, count, position);

方法的输入参数position表示从position处开始向目标文件写入数据，count表示最多传输的字节数。如果源通道的剩余空间小于 count 个字节，则所传输的字节数要小于请求的字节数。

此外要注意，在SoketChannel的实现中，SocketChannel只会传输此刻准备好的数据（可能不足count字节）。因此，SocketChannel可能不会将请求的所有数据(count个字节)全部传输到FileChannel中。

##### transferTo()

**transferTo()方法将数据从FileChannel传输到其他的channel中**。

下面是一个简单的例子：

RandomAccessFile fromFile = **new** RandomAccessFile("G:\\a.txt", "rw");

RandomAccessFile toFile = **new** RandomAccessFile("G:\\b.txt","rw");

FileChannel formChannel = fromFile.getChannel();

FileChannel toChannel = toFile.getChannel();

**long** position = 0;

**long** count = formChannel.size();

formChannel.transferTo(position, count, toChannel);

是不是发现这个例子和前面那个例子特别相似？除了调用方法的FileChannel对象不一样外，其他的都一样。  
上面所说的关于**SocketChannel的问题在transferTo()方法中同样存在。SocketChannel会一直传输数据直到目标buffer被填满**。

#### （六） Selector

Selector（选择器）是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件做好准备的组件。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。

下面是本文所涉及到的主题列表：

1. [为什么使用Selector?](http://ifeve.com/selectors/#Why)
2. [Selector的创建](http://ifeve.com/selectors/#Creating)
3. [向Selector注册通道](http://ifeve.com/selectors/#Registering)
4. [SelectionKey](http://ifeve.com/selectors/#SelectionKey)
5. [通过Selector选择通道](http://ifeve.com/selectors/#Selecting)
6. [wakeUp()](http://ifeve.com/selectors/#wakeUp)
7. [close()](http://ifeve.com/selectors/#close)
8. [完整的示例](http://ifeve.com/selectors/#Full)

##### 为什么使用Selector?

仅用单个线程来处理多个Channels的好处是，只需要更少的线程来处理通道。事实上，可以只用一个线程处理所有的通道。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系统的一些资源（如内存）。因此，使用的线程越少越好。

但是，需要记住，现代的操作系统和CPU在多任务方面表现的越来越好，所以多线程的开销随着时间的推移，变得越来越小了。实际上，如果一个CPU有多个内核，不使用多任务可能是在浪费CPU能力。不管怎么说，关于那种设计的讨论应该放在另一篇不同的文章中。在这里，只要知道使用Selector能够处理多个通道就足够了。

下面是单线程使用一个Selector处理3个channel的示例图：

##### Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

|  |
| --- |
| **Selector selector = Selector.open();** |

|  |
| --- |
|  |

##### 向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。通过SelectableChannel.register()方法来实现，如下：

|  |
| --- |
| **channel.configureBlocking(false);** |
| **SelectionKey key = channel.register(selector, Selectionkey.OP\_READ);** |

与Selector一起使用时，Channel必须处于**非阻塞**模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件：

**Connect**

**Accept**

**Read**

**Write**

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

**SelectionKey.OP\_CONNECT**

**SelectionKey.OP\_ACCEPT**

**SelectionKey.OP\_READ**

**SelectionKey.OP\_WRITE**

如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

|  |
| --- |
| **int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;** |
|  |

在下面还会继续提到interest集合。

##### SelectionKey

在上一小节中，当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些你感兴趣的属性：

**interest集合**

**ready集合**

**Channel**

**Selector**

附加的对象（可选）

下面我会描述这些属性。

###### interest集合

就像[向Selector注册通道](http://ifeve.com/selectors/#Registering)一节中所描述的，interest集合是你所选择的感兴趣的事件集合。可以通过SelectionKey读写interest集合，像这样：

|  |
| --- |
| **int interestSet = selectionKey.interestOps();** |
| **boolean isInterestedInAccept = (interestSet & SelectionKey.OP\_ACCEPT)**  **== SelectionKey.OP\_ACCEPT;** |
| **boolean isInterestedInConnect = interestSet & SelectionKey.OP\_CONNECT;** |
| **boolean isInterestedInRead = interestSet & SelectionKey.OP\_READ;** |
| **boolean isInterestedInWrite = interestSet & SelectionKey.OP\_WRITE;** |

可以看到，用“位与”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的事件是否在interest 集合中。

###### ready集合

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。Selection将在下一小节进行解释。可以这样访问ready集合：

|  |
| --- |
| **int readySet = selectionKey.readyOps();** |

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：

|  |
| --- |
| **selectionKey.isAcceptable();** |
| **selectionKey.isConnectable();** |
| **selectionKey.isReadable();** |
| **selectionKey.isWritable();** |

###### Channel + Selector

从SelectionKey访问Channel和Selector很简单。如下：

|  |
| --- |
| **Channel channel = selectionKey.channel();** |
| **Selector selector = selectionKey.selector();** |

|  |  |  |
| --- | --- | --- |
|  |  | |
|  |  |

###### 附加的对象

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

|  |
| --- |
| **selectionKey.attach(theObject);** |
| **Object arrachedObj = selectionKey.attachment();** |

还可以在用register()方法向Selector注册Channel的时候附加对象。如：

|  |
| --- |
| **selectionKey key = channel.register(selector,** |
| **SelectionKey.OP\_READ,theObject);** |

|  |  |
| --- | --- |
|  |  |

##### 通过Selector选择通道

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道。换句话说，如果你对“读就绪”的通道感兴趣，select()方法会返回读事件已经就绪的那些通道。

下面是select()方法：

**int select()**

**int select(long timeout)**

**int selectNow()**

select()阻塞到至少有一个通道在你注册的事件上就绪了。

select(long timeout)和select()一样，除了最长会阻塞timeout毫秒(参数)。

selectNow()不会阻塞，不管什么通道就绪都立刻返回*（译者注：此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。）。*

select()方法返回的int值表示有多少通道已经就绪。亦即，自上次调用select()方法后有多少通道变成就绪状态。如果调用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调用select()方法，如果另一个通道就绪了，它会再次返回1。如果对第一个就绪的channel没有做任何操作，现在就有两个就绪的通道，但在每次select()方法调用之间，只有一个通道就绪了。

###### （Current—————>>>>>>）selectedKeys()

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。如下所示：

|  |  |
| --- | --- |
| 1 | Set selectedKeys = selector.selectedKeys(); |

当像Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。可以通过SelectionKey的selectedKeySet()方法访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。如下：

|  |  |
| --- | --- |
| 01 | Set selectedKeys = selector.selectedKeys(); |
| 02 | Iterator keyIterator = selectedKeys.iterator(); | |

|  |  |
| --- | --- |
| 03 | while(keyIterator.hasNext()) { |
| 04 | SelectionKey key = keyIterator.next(); | |

|  |  |
| --- | --- |
| 05 | if(key.isAcceptable()) { |
| 06 | // a connection was accepted by a ServerSocketChannel. | |

|  |  |
| --- | --- |
| 07 | } else if (key.isConnectable()) { |
| 08 | // a connection was established with a remote server. | |

|  |  |
| --- | --- |
| 09 | } else if (key.isReadable()) { |
| 10 | // a channel is ready for reading | |

|  |  |
| --- | --- |
| 11 | } else if (key.isWritable()) { |
| 12 | // a channel is ready for writing | |

|  |  |
| --- | --- |
| 13 | } |
| 14 | keyIterator.remove(); | |

|  |  |
| --- | --- |
| 15 | } |

这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

###### wakeUp()

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select()方法返回。只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在select()方法上的线程会立马返回。

如果有其它线程调用了wakeup()方法，但当前没有线程阻塞在select()方法上，下个调用select()方法的线程会立即“醒来（wake up）”。

###### close()

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭。

###### 完整的示例

这里有一个完整的示例，打开一个Selector，注册一个通道注册到这个Selector上(通道的初始化过程略去),然后持续监控这个Selector的四种事件（接受，连接，读，写）是否就绪。

[view source](http://ifeve.com/selectors/#viewSource)

[print](http://ifeve.com/selectors/#printSource)[?](http://ifeve.com/selectors/#about)

|  |  |  |
| --- | --- | --- |
| 01 | Selector selector = Selector.open(); | |
| 02 | channel.configureBlocking(false); |

|  |  |  |
| --- | --- | --- |
| 03 | SelectionKey key = channel.register(selector, SelectionKey.OP\_READ); | |
| 04 | while(true) { |

|  |  |  |
| --- | --- | --- |
| 05 | int readyChannels = selector.select(); | |
| 06 | if(readyChannels == 0) continue; |

|  |  |
| --- | --- |
| 07 | Set selectedKeys = selector.selectedKeys(); |
| 08 | Iterator keyIterator = selectedKeys.iterator(); | |

|  |  |
| --- | --- |
| 09 | while(keyIterator.hasNext()) { |
| 10 | SelectionKey key = keyIterator.next(); | |

|  |  |
| --- | --- |
| 11 | if(key.isAcceptable()) { |
| 12 | // a connection was accepted by a ServerSocketChannel. | |

|  |  |
| --- | --- |
| 13 | } else if (key.isConnectable()) { |
| 14 | // a connection was established with a remote server. | |

|  |  |
| --- | --- |
| 15 | } else if (key.isReadable()) { |
| 16 | // a channel is ready for reading | |

|  |  |
| --- | --- |
| 17 | } else if (key.isWritable()) { |
| 18 | // a channel is ready for writing | |

|  |  |
| --- | --- |
| 19 | } |
| 20 | keyIterator.remove(); | |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 | } |

#### （七） FileChannel

Java NIO中的FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。

FileChannel无法设置为非阻塞模式，它总是运行在阻塞模式下。

##### 打开FileChannel

在使用FileChannel之前，必须先打开它。但是，我们无法直接打开一个FileChannel，需要通过使用一个InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。下面是通过RandomAccessFile打开FileChannel的示例：

|  |  |  |
| --- | --- | --- |
| 1 | RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw"); | |
| 2 | FileChannel inChannel = aFile.getChannel(); |

##### 从FileChannel读取数据

调用多个read()方法之一从FileChannel中读取数据。如：

|  |  |  |
| --- | --- | --- |
| 1 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 2 | int bytesRead = inChannel.read(buf); |

首先，分配一个Buffer。从FileChannel中读取的数据将被读到Buffer中。

然后，调用FileChannel.read()方法。该方法将数据从FileChannel读取到Buffer中。read()方法返回的int值表示了有多少字节被读到了Buffer中。如果返回-1，表示到了文件末尾。

##### 向FileChannel写数据

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer。如：

|  |  |  |
| --- | --- | --- |
| 01 | String newData = "New String to write to file..." + System.currentTimeMillis(); | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 04 | buf.clear(); |

|  |  |  |
| --- | --- | --- |
| 05 | buf.put(newData.getBytes()); | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | buf.flip(); | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | while(buf.hasRemaining()) { |
| 10 | channel.write(buf); |

|  |  |
| --- | --- |
| 11 | } |

注意FileChannel.write()是在while循环中调用的。因为无法保证write()方法一次能向FileChannel写入多少字节，因此需要重复调用write()方法，直到Buffer中已经没有尚未写入通道的字节。

##### 关闭FileChannel

用完FileChannel后必须将其关闭。如：

|  |  |
| --- | --- |
| 1 | channel.close(); |

##### FileChannel的position方法

有时可能需要在FileChannel的某个特定位置进行数据的读/写操作。可以通过调用position()方法获取FileChannel的当前位置。

也可以通过调用position(long pos)方法设置FileChannel的当前位置。

这里有两个例子:

|  |  |  |
| --- | --- | --- |
| 1 | long pos = channel.position(); | |
| 2 | channel.position(pos +123); |

如果将位置设置在文件结束符之后，然后试图从文件通道中读取数据，读方法将返回-1 —— 文件结束标志。

如果将位置设置在文件结束符之后，然后向通道中写数据，文件将撑大到当前位置并写入数据。这可能导致“文件空洞”，磁盘上物理文件中写入的数据间有空隙。

##### FileChannel的size方法

FileChannel实例的size()方法将返回该实例所关联文件的大小。如:

|  |  |
| --- | --- |
| 1 | long fileSize = channel.size(); |

##### FileChannel的truncate方法

可以使用FileChannel.truncate()方法截取一个文件。截取文件时，文件将中指定长度后面的部分将被删除。如：

|  |  |
| --- | --- |
| 1 | channel.truncate(1024); |

这个例子截取文件的前1024个字节。

##### FileChannel的force方法

FileChannel.force()方法将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，操作系统会将数据缓存在内存中，所以无法保证写入到FileChannel里的数据一定会即时写到磁盘上。要保证这一点，需要调用force()方法。

force()方法有一个boolean类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

下面的例子同时将文件数据和元数据强制写到磁盘上：

|  |  |
| --- | --- |
| 1 | channel.force(true); |
|  |  |

#### （八） SocketChannel

Java NIO中的SocketChannel是一个连接到TCP网络套接字的通道。可以通过以下2种方式创建SocketChannel：

1. 打开一个SocketChannel并连接到互联网上的某台服务器。
2. 一个新连接到达ServerSocketChannel时，会创建一个SocketChannel。

##### 打开 SocketChannel

下面是SocketChannel的打开方式：

|  |  |
| --- | --- |
| 1 | SocketChannel socketChannel = SocketChannel.open(); |
| 2 | socketChannel.connect(new InetSocketAddress("[http://jenkov.com](http://jenkov.com/)", 80)); | |

##### 关闭 SocketChannel

当用完SocketChannel之后调用SocketChannel.close()关闭SocketChannel：

|  |  |
| --- | --- |
| 1 | socketChannel.close(); |

##### 从 SocketChannel 读取数据

要从SocketChannel中读取数据，调用一个read()的方法之一。以下是例子：

|  |  |
| --- | --- |
| 1 | ByteBuffer buf = ByteBuffer.allocate(48); |
| 2 | int bytesRead = socketChannel.read(buf); |

首先，分配一个Buffer。从SocketChannel读取到的数据将会放到这个Buffer中。

然后，调用SocketChannel.read()。该方法将数据从SocketChannel 读到Buffer中。read()方法返回的int值表示读了多少字节进Buffer里。如果返回的是-1，表示已经读到了流的末尾（连接关闭了）。

##### 写入 SocketChannel

写数据到SocketChannel用的是SocketChannel.write()方法，该方法以一个Buffer作为参数。示例如下：

|  |  |  |
| --- | --- | --- |
| 01 | String newData = "New String to write to file..." + System.currentTimeMillis(); | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 04 | buf.clear(); |

|  |  |  |
| --- | --- | --- |
| 05 | buf.put(newData.getBytes()); | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | buf.flip(); | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | while(buf.hasRemaining()) { |
| 10 | channel.write(buf); |

|  |  |
| --- | --- |
| 11 | } |

注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证能写多少字节到SocketChannel。所以，我们重复调用write()直到Buffer没有要写的字节为止。

##### 非阻塞模式

可以设置 SocketChannel 为非阻塞模式（non-blocking mode）.设置之后，就可以在异步模式下调用connect(), read() 和write()了。

###### connect()

如果SocketChannel在非阻塞模式下，此时调用connect()，该方法可能在连接建立之前就返回了。为了**确定连接是否建立**，可以调用**finishConnect()**的方法。像这样：

|  |  |
| --- | --- |
| 1 | socketChannel.configureBlocking(false); |
| 2 | socketChannel.connect(new InetSocketAddress("[http://jenkov.com](http://jenkov.com/)", 80)); | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | while(! socketChannel.finishConnect() ){ | |

|  |  |  |
| --- | --- | --- |
| 5 | //wait, or do something else... | |
| 6 | } |

###### write()

非阻塞模式下，write()方法在尚未写出任何内容时可能就返回了。所以需要在循环中调用write()。前面已经有例子了，这里就不赘述了。

###### read()

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的int返回值，它会告诉你读取了多少字节。

##### 非阻塞模式与选择器

非阻塞模式与选择器搭配会工作的更好，通过将一或多个SocketChannel注册到Selector，可以询问选择器哪个通道已经准备好了读取，写入等。Selector与SocketChannel的搭配使用会在后面详讲。

#### （九） ServerSocketChannel

Java NIO中的 ServerSocketChannel 是一个可以监听新进来的TCP连接的通道, 就像标准IO中的ServerSocket一样。ServerSocketChannel类在 java.nio.channels包中。

这里有个例子：

|  |  |  |
| --- | --- | --- |
| 01 | ServerSocketChannel serverSocketChannel = ServerSocketChannel.open(); | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | serverSocketChannel.socket().bind(new InetSocketAddress(9999)); | |
| 04 |  |

|  |  |
| --- | --- |
| 05 | while(true){ |
| 06 | SocketChannel socketChannel = | |

|  |  |  |
| --- | --- | --- |
| 07 | serverSocketChannel.accept(); | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | //do something with socketChannel... | |
| 10 | } |

##### 打开 ServerSocketChannel

通过调用 ServerSocketChannel.open() 方法来打开ServerSocketChannel.如：

|  |  |
| --- | --- |
| 1 | ServerSocketChannel serverSocketChannel = ServerSocketChannel.open(); |

##### 关闭 ServerSocketChannel

通过调用ServerSocketChannel.close() 方法来关闭ServerSocketChannel. 如：

|  |  |
| --- | --- |
| 1 | serverSocketChannel.close(); |

##### 监听新进来的连接

通过 ServerSocketChannel.accept() 方法监听新进来的连接。当 accept()方法返回的时候,它返回一个包含新进来的连接的 SocketChannel。因此, accept()方法会一直阻塞到有新连接到达。

通常不会仅仅只监听一个连接,在while循环中调用 accept()方法. 如下面的例子：

|  |  |
| --- | --- |
| 1 | while(true){ |
| 2 | SocketChannel socketChannel = | |

|  |  |  |
| --- | --- | --- |
| 3 | serverSocketChannel.accept(); | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | //do something with socketChannel... | |
| 6 | } |

当然,也可以在while循环中使用除了true以外的其它退出准则。

##### 非阻塞模式

ServerSocketChannel可以设置成非阻塞模式。在非阻塞模式下，accept() 方法会立刻返回，如果还没有新进来的连接,返回的将是null。 因此，需要检查返回的SocketChannel是否是null.如：

[view source](http://ifeve.com/server-socket-channel/#viewSource)

[print](http://ifeve.com/server-socket-channel/#printSource)[?](http://ifeve.com/server-socket-channel/#about)

|  |  |  |
| --- | --- | --- |
| 01 | ServerSocketChannel serverSocketChannel = ServerSocketChannel.open(); | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | serverSocketChannel.socket().bind(new InetSocketAddress(9999)); | |
| 04 | serverSocketChannel.configureBlocking(false); |

|  |  |
| --- | --- |
| 05 |  |
| 06 | while(true){ | |

|  |  |
| --- | --- |
| 07 | SocketChannel socketChannel = |
| 08 | serverSocketChannel.accept(); | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | if(socketChannel != null){ | |

|  |  |  |
| --- | --- | --- |
| 11 | //do something with socketChannel... | |
| 12 | } |

|  |  |
| --- | --- |
| 13 | } |

#### （十） Java NIO DatagramChannel

Java NIO中的DatagramChannel是一个能收发UDP包的通道。因为UDP是无连接的网络协议，所以不能像其它通道那样读取和写入。它发送和接收的是数据包。

##### 打开 DatagramChannel

下面是 DatagramChannel 的打开方式：

|  |  |
| --- | --- |
| 1 | DatagramChannel channel = DatagramChannel.open(); |
| 2 | channel.socket().bind(new InetSocketAddress(9999)); | |

这个例子打开的 DatagramChannel可以在UDP端口9999上接收数据包。

##### 接收数据 receive()

通过**receive()**方法从DatagramChannel接收数据，如：

|  |  |  |
| --- | --- | --- |
| 1 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 2 | buf.clear(); |

|  |  |
| --- | --- |
| 3 | channel.receive(buf); |

receive()方法会将接收到的数据包内容复制到指定的Buffer. 如果Buffer容不下收到的数据，多出的数据将被丢弃。

##### 发送数据send()

通过send()方法从DatagramChannel发送数据，如:

|  |  |  |
| --- | --- | --- |
| 1 | String newData = "New String to write to file..." + System.currentTimeMillis(); | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 4 | buf.clear(); |

|  |  |  |
| --- | --- | --- |
| 5 | buf.put(newData.getBytes()); | |
| 6 | buf.flip(); |

|  |  |
| --- | --- |
| 7 |  |
| 8 | int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80)); | |

这个例子发送一串字符到”jenkov.com”服务器的UDP端口80。 因为服务端并没有监控这个端口，所以什么也不会发生。也不会通知你发出的数据包是否已收到，因为UDP在数据传送方面没有任何保证。

##### 连接到特定的地址connect()

可以将DatagramChannel“连接”到网络中的特定地址的。由于UDP是无连接的，连接到特定地址并不会像TCP通道那样创建一个真正的连接。而是锁住DatagramChannel ，让其只能从特定地址收发数据。

这里有个例子:

|  |  |
| --- | --- |
| 1 | **channel.connect(new InetSocketAddress("jenkov.com", 80));** |

当连接后，也可以使用read()和write()方法，就像在用传统的通道一样。只是在数据传送方面没有任何保证。这里有几个例子：

[view source](http://ifeve.com/datagram-channel/#viewSource)

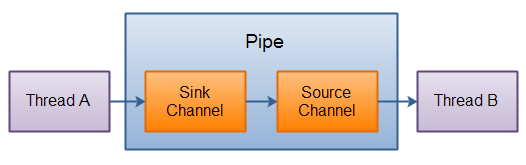
[print](http://ifeve.com/datagram-channel/#printSource)[?](http://ifeve.com/datagram-channel/#about)

|  |  |
| --- | --- |
| 1 | int bytesRead = channel.read(buf); |
| 2 | int bytesWritten = channel.write(but); | |

#### （十一） Pipe

Java NIO 管道是2个线程之间的单向数据连接。Pipe有一个source通道和一个sink通道。数据会被写到sink通道，从source通道读取。

这里是Pipe原理的图示：



##### 创建管道

通过Pipe.open()方法打开管道。例如：

|  |  |
| --- | --- |
| 1 | Pipe pipe = Pipe.open(); |

##### 向管道写数据

要向管道写数据，需要访问sink通道。像这样：

|  |  |
| --- | --- |
| 1 | Pipe.SinkChannel sinkChannel = pipe.sink(); |

通过调用SinkChannel的write()方法，将数据写入SinkChannel,像这样：

|  |  |  |
| --- | --- | --- |
| 01 | String newData = "New String to write to file..." + System.currentTimeMillis(); | |
| 02 | ByteBuffer buf = ByteBuffer.allocate(48); |

|  |  |
| --- | --- |
| 03 | buf.clear(); |
| 04 | buf.put(newData.getBytes()); | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | buf.flip(); | |

|  |  |
| --- | --- |
| 07 |  |
| 08 | while(buf.hasRemaining()) { | |

|  |  |  |
| --- | --- | --- |
| 09 | sinkChannel.write(buf); | |
| 10 | } |

##### 从管道读取数据

从读取管道的数据，需要访问source通道，像这样：

|  |  |
| --- | --- |
| 1 | Pipe.SourceChannel sourceChannel = pipe.source(); |

调用source通道的read()方法来读取数据，像这样：

|  |  |  |
| --- | --- | --- |
| 1 | ByteBuffer buf = ByteBuffer.allocate(48); | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | int bytesRead = sourceChannel.read(buf); |

read()方法返回的int值会告诉我们多少字节被读进了缓冲区。

#### （十二） Java NIO与IO

当学习了Java NIO和IO的API后，一个问题马上涌入脑海：

我应该何时使用IO，何时使用NIO呢？在本文中，我会尽量清晰地解析Java NIO和IO的差异、它们的使用场景，以及它们如何影响您的代码设计。

##### Java NIO和IO的主要区别

下表总结了Java NIO和IO之间的主要差别，我会更详细地描述表中每部分的差异。

**IO**                **NIO**

面向流            面向缓冲

阻塞IO           非阻塞IO

无 选择器

##### ****面向流与面向缓冲****

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

##### ****阻塞与非阻塞IO****

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

##### ****选择器（****Selectors****）****

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

##### ****NIO和IO如何影响应用程序的设计****

无论您选择IO或NIO工具箱，可能会影响您应用程序设计的以下几个方面：

1. 对NIO或IO类的API调用。
2. 数据处理。
3. 用来处理数据的线程数。

###### ****API调用****

当然，使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

###### ****数据处理****

使用纯粹的NIO设计相较IO设计，数据处理也受到影响。

在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

Name: Anna

Age: 25

Email: anna@mailserver.com

Phone: 1234567890

该文本行的流可以这样处理：  
InputStream input = … ; // get the InputStream from the client socket

|  |  |  |
| --- | --- | --- |
| 1 | BufferedReader reader = new BufferedReader(new InputStreamReader(input)); | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | String nameLine   = reader.readLine(); |
| 4 | String ageLine    = reader.readLine(); | |

|  |  |
| --- | --- |
| 5 | String emailLine  = reader.readLine(); |
| 6 | String phoneLine  = reader.readLine(); |

请注意处理状态由程序执行多久决定。换句话说，一旦reader.readLine()方法返回，你就知道肯定文本行就已读完， readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个readline()调用返回的时候，你知道这行包含年龄等。 正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。下图也说明了这条原则：（**Java IO: 从一个阻塞的流中读数据**） 而一个NIO的实现会有所不同，下面是一个简单的例子：

|  |  |  |
| --- | --- | --- |
| 1 | ByteBuffer buffer = ByteBuffer.allocate(48); | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | int bytesRead = inChannel.read(buffer); |

注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。  
假设第一次 read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

|  |  |  |
| --- | --- | --- |
| 1 | ByteBuffer buffer = ByteBuffer.allocate(48); | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | int bytesRead = inChannel.read(buffer); | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | while(! bufferFull(bytesRead) ) { | |
| 6 |  |

|  |  |  |
| --- | --- | --- |
| 7 | bytesRead = inChannel.read(buffer); | |
| 8 |  |

|  |  |
| --- | --- |
| 9 | } |

bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull（）方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

如果缓冲区已满，它可以被处理。如果它不满，并且在你的实际案例中有意义，你或许能处理其中的部分数据。但是许多情况下并非如此。下图展示了“缓冲区数据循环就绪”：

**Java NIO:从一个通道里读数据，直到所有的数据都读到缓冲区里.**

**3) 用来处理数据的线程数**

NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，实现NIO的服务器可能是一个优势。同样，如果你需要维持许多打开的连接到其他计算机上，如P2P网络中，使用一个单独的线程来管理你所有出站连接，可能是一个优势。一个线程多个连接的设计方案如下图所示：

**Java NIO: 单线程管理多个连接**

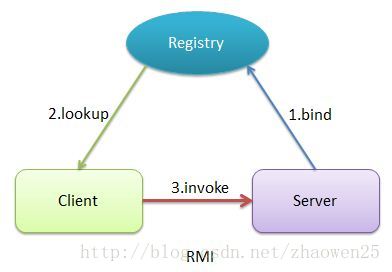
如果你有少量的连接使用非常高的带宽，一次发送大量的数据，也许典型的IO服务器实现可能非常契合。下图说明了一个典型的IO服务器设计：

**Java IO: 一个典型的IO服务器设计- 一个连接通过一个线程处理.**

### 3.RPC

**RPC**是远程过程调用的简称，广泛应用在大规模分布式应用中，作用是有助于系统的垂直拆分，使系统更易拓展。Java中的RPC框架比较多，各有特色，广泛使用的有RMI、Hessian、Dubbo等。RPC还有一个特点就是能够跨语言，本文只以JAVA语言里的RPC为例。

对于RPC有一个逻辑关系图，以RMI为例：



其他的框架结构也类似，区别在于对象的序列化方法，传输对象的通讯协议，以及注册中心的管理与failover设计（利用zookeeper）。

客户端和服务端可以运行在不同的JVM中，Client只需要引入接口，接口的实现以及运行时需要的数据都在Server端，RPC的主要依赖技术是序列化、反序列化和传输协议，JAVA里对应的就是对象的序列化、反序列化以及序列化后数据的传输。RMI的序列化和反序列化是JAVA自带的，Hessian里的序列化和反序列化是私有的，传输协议则是HTTP，Dubbo的序列化可以多种选择，一般使用Hessian的序列化协议，传输则是TCP协议，使用了高性能的NIO框架Netty。对于序列化，我还了解一些，像Google的ProBuffer、JBoss Marshalling和Apache Thrift等，之前有写一篇介绍ProBuffer的博文

#### ****1、RMI（远程方法调用）****

JAVA自带的远程方法调用工具，不过有一定的局限性，毕竟是JAVA语言最开始时的设计，后来很多框架的原理都基于RMI，RMI的使用如下：

**对外接口**

**[java]**

1. **public** **interface** IService **extends** Remote {
2. **public** String queryName(String no) **throws** RemoteException;
4. }

**服务实现**

**[java]**

1. **import** java.rmi.RemoteException;
2. **import** java.rmi.server.UnicastRemoteObject;
4. // 服务实现
5. **public** **class** ServiceImpl **extends** UnicastRemoteObject **implements** IService {
7. /\*\*
8. \*/
9. **private** **static** **final** **long** serialVersionUID = 682805210518738166L;
11. /\*\*
12. \* @throws RemoteException
13. \*/
14. **protected** ServiceImpl() **throws** RemoteException {
15. **super**();
16. }
18. /\* (non-Javadoc)
19. \*
20. \*/
21. @Override
22. **public** String queryName(String no) **throws** RemoteException {
23. // 方法的具体实现
24. System.out.println("hello" + no);
25. **return** String.valueOf(System.currentTimeMillis());
26. }
28. }

**RMI客户端**

**[java]**

1. **import** java.rmi.AccessException;
2. **import** java.rmi.NotBoundException;
3. **import** java.rmi.RemoteException;
4. **import** java.rmi.registry.LocateRegistry;
5. **import** java.rmi.registry.Registry;
7. // RMI客户端
8. **public** **class** Client {
10. **public** **static** **void** main(String[] args) {
11. // 注册管理器
12. Registry registry = **null**;
13. **try** {
14. // 获取服务注册管理器
15. registry = LocateRegistry.getRegistry("127.0.0.1",8088);
16. // 列出所有注册的服务
17. String[] list = registry.list();
18. **for**(String s : list){
19. System.out.println(s);
20. }
21. } **catch** (RemoteException e) {
23. }
24. **try** {
25. // 根据命名获取服务
26. IService server = (IService) registry.lookup("vince");
27. // 调用远程方法
28. String result = server.queryName("ha ha ha ha");
29. // 输出调用结果
30. System.out.println("result from remote : " + result);
31. } **catch** (AccessException e) {
33. } **catch** (RemoteException e) {
35. } **catch** (NotBoundException e) {
37. }
38. }
39. }

**RMI服务端**

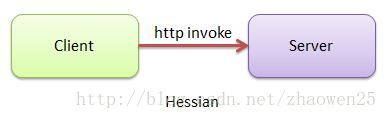
**[java]**

1. **import** java.rmi.RemoteException;
2. **import** java.rmi.registry.LocateRegistry;
3. **import** java.rmi.registry.Registry;
5. // RMI服务端
6. **public** **class** Server {
8. **public** **static** **void** main(String[] args) {
9. // 注册管理器
10. Registry registry = **null**;
11. **try** {
12. // 创建一个服务注册管理器
13. registry = LocateRegistry.createRegistry(8088);
15. } **catch** (RemoteException e) {
17. }
18. **try** {
19. // 创建一个服务
20. ServiceImpl server = **new** ServiceImpl();
21. // 将服务绑定命名
22. registry.rebind("vince", server);
24. System.out.println("bind server");
25. } **catch** (RemoteException e) {
27. }
28. }
29. }

服务注册管理器写在了Server里，当然也可以抽出来单独作为一个服务，在其他一些框架中，往往用Zookeeper充当注册管理角色。

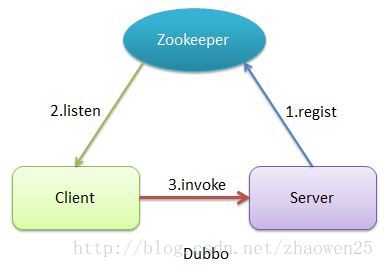
#### ****2、Hessian（基于HTTP的远程方法调用）****

基于HTTP协议传输，在性能方面还不够完美，负载均衡和失效转移依赖于应用的负载均衡器，Hessian的使用则与RMI类似，区别在于淡化了Registry的角色，通过显示的地址调用，利用HessianProxyFactory根据配置的地址create一个代理对象，另外还要引入Hessian的Jar包。



#### ****3、Dubbo（淘宝开源的基于TCP的RPC框架）****

基于Netty的高性能RPC框架，是阿里巴巴开源的，总体原理如下：



在了解[Dubbo](https://github.com/alibaba/dubbo)之前，要先对Zookeeper有深入的理解，当理解了zookeeper后，Dubbo也就了无秘密了。

Dubbo的详细说明在淘宝开源里说的非常详细，在工作中很多生产项目都用了Dubbo，过程中也发现了很多需要注意的地方，尤其是那繁多的配置，设置不当都会让人烦脑，最好能再基于现有开源的Dubbo再定制优化一下。

## 二十二、并发库、多线程、JVM调度、序列化

### \*并发库(java.util.concurrent.\*)

[java](eclipse-javadoc:%E2%98%82=java-base/E:%5C/Jdk%5C/java8%5C/jre8%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=java-base/E:%5C/Jdk%5C/java8%5C/jre8%5C/lib%5C/rt.jar%3Cjava.util).[concurrent](eclipse-javadoc:%E2%98%82=java-base/E:%5C/Jdk%5C/java8%5C/jre8%5C/lib%5C/rt.jar%3Cjava.util.concurrent).[atomic](eclipse-javadoc:%E2%98%82=java-base/E:%5C/Jdk%5C/java8%5C/jre8%5C/lib%5C/rt.jar%3Cjava.util.concurrent.atomic).AtomicInteger

#### 1.AtomicInteger

**package** com.xgb.java.base.concurrent;

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** TestCOncurrent {

//volatile 关键字 (能使变量具有可见性和有序性)

**private** **static** **volatile** **int** *num* = 1000;//内存屏障和内存可见性

//同步代价太大

/\* private static synchronized void add() {

num ++;

}\*/

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

AtomicInteger atomicIntegr = **new** AtomicInteger(*num*);

**for** (**int** i = 0; i < 1000; i++) {

**new** Thread() {

**public** **void** run() {

// add();

// atomicIntegr.getAndIncrement();

atomicIntegr.decrementAndGet();

};

}.start();

}

Thread.*sleep*(1000);

System.***out***.println("num: " + atomicIntegr.get());

}

}

### 1.多线程

#### （1）进程和线程的概述

##### 1.定义

* 进程就是操作系统对资源进行分配调度的一个独立单元。
* 线程就进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，线程自己基本上不拥有资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可以与同属于一个进程的其他线程共享进程所拥有的全部资源。

##### 2.关系

一个线程可以创建和销毁另一个线程，同一个进程内的线程并行执行，并且共享进程的资源，但拥有自己的栈空间和执行序列。

##### 3.区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。**进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径**。**线程有自己的堆栈和局部变量**，**但线程之间没有单独的地址空间**，**一个线程死掉就等于整个进程死掉**，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。**但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。**

**1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.**

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。**但是线程不能够独立执行，**必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。**这就是进程和线程的重要区别。**

##### 4.优缺点

线程和进程在使用上各有优缺点：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。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

#### （2）线程安全问题

这里以简单的卖票程序为例开始本文的讲解，通过运行该卖票程序可以发现有可能打印出0、-1、-2等错票，这就表明多线程的运行出现了安全问题。产生的原因可归纳为两点：

1. 线程任务中有处理到共享的数据。
2. 线程任务中有多条对共享数据的操作。一个线程在操作共享数据的过程中，其他线程参与了运算，造成了数据的错误。

解决办法：只要能保证多条操作共享数据的代码在某一时间段被一条线程所执行，在执行期间不允许其他线程参与运算。   
Java对于多线程的安全问题提供了专业的解决方式——就是同步代码块。   
**注意：线程安全问题在理想状态下，不容易出现，但一旦出现对软件的影响是非常大的。**

#### （3）同步（synchronized）

synchronized(对象) {

需要被同步的代码

}

**同步的前提：**

1. **必须要有两个或者两个以上的线程**
2. **必须是多个线程使用同一个锁**

同步的好处：解决了多线程的安全问题。   
同步的弊端：多个线程需要判断锁，较为消耗资源。

如何找到问题：

1. 明确哪些代码是多线程运行代码
2. 明确共享数据
3. 明确多线程运行代码中哪些语句是操作共享数据的

#### （4）多线程间的通信

本文继续接着讲解多线程，这儿会涉及到多线程的高级部分——**多线程间的通信。 多线程间的通信其实就是多个线程都在处理同一个资源，但是处理的任务却不一样**。最经典的案例就是生产者和消费者的案例了，下面我就来演示该案例。

//资源类

**package** com.ws.multi.Lock;

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** Resource {

**private** **int** count;//

**private** **boolean** flag;

**public** **synchronized** **void** set(){

**while**(flag){//判断flag为true时 wait

**try** {

wait();//线程等待不可以生产

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

count ++;

flag = **true**;//更改标记为true 表示已经生产完成可以消费了

System.***out***.println(Thread.*currentThread*().getName() + "--生产者--" + count);

notifyAll();//唤醒消费者，通知可以消费了

}

**public** **synchronized** **void** get(){

**while**(!flag){

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

flag = **false**;//更改标记为false 表示已经消费完成可以生产了

System.***out***.println(Thread.*currentThread*().getName() + "--消费者--" + count + "\n");

notifyAll();//唤醒生产者,通知可以生产了

}

/\*\*

\* 使用Lock锁和Condition监视锁来实现多线程的生产者与消费者模式

\*/

// private Lock lock = new ReentrantLock();

// private Condition product = lock.newCondition();

// private Condition customer = lock.newCondition();

//

// public void set(){

// lock.lock();

// try {

// while(flag){// 判断标记为true，就执行wait()等待。为false，就生产。

// try {

// product.await();//让生产wait

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

// }

// count ++;

// //生产完毕改变flag为true 不能生产

// flag = true;

// System.out.println(Thread.currentThread().getName() + "--生产者完成--" + count);

// //生产完成，唤醒一个消费者来消费

// customer.signal();

//

// } finally {

// lock.unlock();

// }

// }

//

// public void get(){

// lock.lock();

// try {

// while(!flag){//判断标记flag为false 表示已经生产完了，等待消费

// try {

// customer.await();

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

// }

// //消费完成，将flag标记为false表示可以生产了

// flag = false;

// System.out.println(Thread.currentThread().getName() + "--消费者完成--" + count + "\n");

//

// //消费完成，唤醒一个生产者生产

// product.signal();

//

// } finally {

// lock.unlock();

// }

// }

}

//生产者

**package** com.ws.multi.Lock;

**public** **class** Product **implements** Runnable{

**private** Resource resource;

**public** Product(Resource resource){

**this**.resource = resource;

}

**public** **void** run() {

**while**(**true**){

//模拟生产耗时

// try {

// Thread.sleep(300);

// } catch (InterruptedException e) {

// e.printStackTrace();

// }

resource.set();

}

}

}

//消费者

**package** com.ws.multi.Lock;

**public** **class** Customer **implements** Runnable{

**private** Resource resource;

**public** Customer(Resource resource){

**this**.resource = resource;

}

**public** **void** run() {

**while**(**true**){

resource.get();

}

}

}

//测试类

**package** com.ws.multi.Lock;

**public** **class** TestPCR {

**public** **static** **void** main(String[] args) {

Resource resource = **new** Resource();

Product pro = **new** Product(resource);

Customer cus = **new** Customer(resource);

Thread t1 = **new** Thread(pro);

Thread t2 = **new** Thread(pro);

Thread t3 = **new** Thread(cus);

Thread t4 = **new** Thread(cus);

t1.setName("生产者A");

t2.setName("生产者B");

t3.setName("消费者C");

t4.setName("消费者D");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

#### （5）java多线程经典实例

##### 一、相关知识：

Java多线程程序设计到的知识：

（一）对同一个数量进行操作

（二）对同一个对象进行操作

（三）回调方法使用

（四）线程同步，死锁问题

（五）线程通信

 等等

##### 二、示例一：三个售票窗口同时出售20张票;

程序分析：1.票数要使用同一个静态值

 2.为保证不会出现卖出同一个票数，要java多线程同步锁。

设计思路：1.创建一个站台类Station，继承Thread，重写run方法，在run方法里面执行售票操作！售票要使用同步锁：即有一个站台卖这张票时，其他站台要等这张票卖完！

2.创建主方法调用类

**//站台类继承Thread**

**package** com.ws.multi.anli.ticket;

**public** **class** Station **extends** Thread{

// 通过构造方法给线程名字赋值

**public** Station(String name) {

**super**(name);// 给线程名字赋值

}

// 为了保持票数的一致，票数要静态

**static** **int** *tick* = 20;

// 创建一个静态钥匙

**static** Object *ob* = "aa";//值是任意的

// 重写run方法，实现买票操作

@Override

**public** **void** run() {

**while** (*tick* > 0) {

**synchronized** (*ob*) {// 这个很重要，必须使用一个锁，

// 进去的人会把钥匙拿在手上，出来后才把钥匙拿让出来

**if** (*tick* > 0) {

System.***out***.println(getName() + "卖出了第" + *tick* + "张票");

*tick*--;

} **else** {

System.***out***.println("票卖完了");

}

}

**try** {

*sleep*(1000);//休息一秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**//创建主方法调用类**

**package** com.ws.multi.anli.ticket;

**public** **class** TestStation {

**public** **static** **void** main(String[] args) {

Thread thread1 = **new** Station("窗口1");

Thread thread2 = **new** Station("窗口2");

Thread thread3 = **new** Station("窗口3");

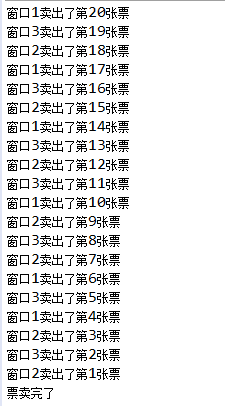
thread1.start();

thread2.start();

thread3.start();

}

}



##### 三．示例二：两个人AB通过一个账户A在柜台取钱和B在ATM机取钱！

//银行类提供柜台取钱和ATM取钱

**package** com.ws.multi.anli.bank;

**public** **class** Bank {

// 假设一个账户有1000块钱

**static** **int** *money* = 1000;

// 柜台Counter取钱的方法

**public** **void** Counter(**int** money) {// 参数是每次取走的钱

Bank.*money* -= money;//取钱后总数减少

System.***out***.println("A取走了" + money + "还剩下" + (Bank.*money*));

}

// ATM取钱的方法

**public** **void** ATM(**int** money) {// 参数是每次取走的钱

Bank.*money* -= money;//取钱后总数减少

System.***out***.println("B取走了" + money + "还剩下" + (Bank.*money*));

}

}

//A

**package** com.ws.multi.anli.bank;

**public** **class** APerson **extends** Thread{

// 创建银行对象

Bank bank;

// 通过构造器传入银行对象，确保两个人进入的是一个银行

**public** APerson(Bank bank) {

**this**.bank = bank;

}

//重写run方法，在里面实现使用柜台取钱

**public** **void** run() {

**while** (Bank.*money* >= 100) {

bank.Counter(100);// 每次取100块

**try** {

*sleep*(100);// 取完休息0.1秒

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

//B

**package** com.ws.multi.anli.bank;

**public** **class** BPerson **extends** Thread{

// 创建银行对象

Bank bank;

// 通过构造器传入银行对象，确保两个人进入的是一个银行

**public** BPerson(Bank bank) {

**this**.bank = bank;

}

//重写run方法，在里面实现使用ATM取钱

**public** **void** run() {

**while**(Bank.*money* >= 200){

bank.ATM(200);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

//测试

**package** com.ws.multi.anli.bank;

**public** **class** TestBank {

**public** **static** **void** main(String[] args) {

// 实力化一个银行对象

Bank bank = **new** Bank();

// 实例化两个人，传入同一个银行的对象

APerson pA = **new** APerson(bank);

BPerson pB = **new** BPerson(bank);

// 两个人开始取钱

pA.start();

pB.start();

}

}

### 2.JVM调度

### 3.序列化

## 二十三、Netty、Dubbo、Mina框架

## 二十三、[RESTful接口设计规范](https://www.cnblogs.com/tommyli/p/3913018.html)、HTTP协议、JPA及WEB Service规范

[RESTful API](http://en.wikipedia.org/wiki/Representational_state_transfer)是目前比较成熟的一套互联网应用程序的API设计理论

* GET（SELECT）：从服务器取出资源（一项或多项）。
* POST（CREATE）：在服务器新建一个资源。
* PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。
* PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。
* DELETE（DELETE）：从服务器删除资源。
* HEAD：获取资源的元数据。
* OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

* ?limit=10：指定返回记录的数量
* ?offset=10：指定返回记录的开始位置。
* ?page=2&per\_page=100：指定第几页，以及每页的记录数。
* ?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。
* ?animal\_type\_id=1：指定筛选条件

### 状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

* 200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
* 201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
* 202 Accepted - [\*]：表示一个请求已经进入后台排队（异步任务）
* 204 NO CONTENT - [DELETE]：用户删除数据成功。
* 400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
* 401 Unauthorized - [\*]：表示用户没有权限（令牌、用户名、密码错误）。
* 403 Forbidden - [\*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
* 404 NOT FOUND - [\*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
* 406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
* 410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
* 422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
* 500 INTERNAL SERVER ERROR - [\*]：服务器发生错误，用户将无法判断发出的请求是否成功。

## intellij idea

## 二十四、设计模式

#### 1.单利设计模式（signle）

懒汉式、饿汉式、双重校验锁、静态加载、枚举类加载。保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**懒汉式、饿汉式在spring IOC中的应用**

在spring IOC中，bean在xml中可以配置为singleton，而且有一个lazy-init属性

lazy-init=true,设置延迟初始化， 在创建容器之后，在第一次从容器获取对象的时候创建单例的对象

如果没有配置或延迟初始化为默认值, 单例的对象会在创建容器的时候创建对象。

##### 懒汉式

/\*\*

\* **@name** 单例模式-懒汉式。

\* 这段代码简单明了，而且使用了懒加载模式，但是却存在致命的问题。

\* 当有多个线程并行调用 getInstance()的时候，就会创建多个实例。

\* 也就是说在多线程下不能正常工作。

\* **@author** xgb

\* **@version** 0.0.1

\* **@data** 2018-5-21

\*/

**public** **class** Signleton {

**private** **static** Signleton *instance* = **null**;

//构造方法私有化

**private** Signleton() { }

//通过静态公共的方式对外提供获取实例

**public** **static** Signleton getInstance() {

//懒汉式，类加载时不创建实例

**if**(*instance* == **null**) {

*instance* = **new** Signleton();

}

**return** *instance*;

}

}

/\*\*

\* **@name** 单例模式-懒汉式-线程安全。

\* 虽然做到了线程安全，并且解决了多实例的问题，但是它并不高效。

\* 因为在任何时候只能有一个线程调用 getInstance()方法。

\* 但是同步操作只需要在第一次调用时才被需要，

\* 即第一次创建单例实例对象时。这就引出了双重检验锁。

\* **@author** xgb

\* **@version** 0.0.2

\* **@data** 2018-5-21

\*/

**public** **class** Signleton {

**private** **static** Signleton *instance*;

**private** Signleton() {}

**public** **static** **synchronized** Signleton getInstacne() {

**if**(*instance* == **null**) {

*instance* = **new** Signleton();

}

**return** *instance*;

}

}

/\*\*

\* **@name** 单例模式-懒汉式（双重校验锁）- 线程安全。

\* 优化了同步方法，提升获取实例的效率。

\* **@author** xgb

\* **@version** 0.0.3

\* **@data** 2018-5-21

\*/

**public** **class** Signleton {

//声明volatile保证编译器不进行优化

**private** **volatile** **static** Signleton *instance*;

**private** Signleton() {}

**public** **static** Signleton getInstacne() {

**if**(*instance* == **null**) {

**synchronized**(Signleton.**class**) {

**if**(*instance* != **null**) {

*instance* = **new** Signleton();

}

}

}

**return** *instance*;

}

}

##### 饿汉式

/\*\*

\* **@name** 单例模式-饿汉式-线程安全。

\* 这种方法非常简单，因为单例的实例被声明成 static和 final变量了，

\* 在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。

\* 这种写法如果完美的话，就没必要在啰嗦那么多双检锁的问题了。缺点是它不是一种懒加载模式（lazy initialization），

\* 单例会在加载类后一开始就被初始化，即使客户端没有调用 getInstance()方法。

\* 饿汉式的创建方式在一些场景中将无法使用：譬如 Singleton 实例的创建是依赖参数或者配置文件的，

\* 在 getInstance()之前必须调用某个方法设置参数给它，那样这种单例写法就无法使用了。

\* **@author** xgb

\* **@version** 0.0.1

\* **@data** 2018-5-21

\*/

**public** **class** Signleton {

//类加载时就初始化

**private** **static** **final** Signleton ***instance*** = **new** Signleton();

**private** Signleton() {}

**public** **static** Signleton getInstance() {

**return** ***instance***;

}

}

##### 静态内部类

/\*\*

\* **@name** 单例模式-静态内部类。

\* 这种写法仍然使用JVM本身机制保证了线程安全问题；由于 SingletonHolder是私有的，

\* 除了 getInstance()之外没有办法访问它，因此它是懒汉式的；同时读取实例的时候不会进行同步，

\* 没有性能缺陷；也不依赖 JDK版本。

\* **@author** xgb

\* **@version** 0.0.1

\* **@data** 2018-5-21

\*/

**public** **class** Signleton {

**private** **static** **class** SignletonHolder{

**private** **static** **final** Signleton ***INSTANCE*** = **new** Signleton();

}

**private** Signleton() {}

**public** **static** **final** Signleton getInstance() {

**return** SignletonHolder.***INSTANCE***;

}

}

#### 2.工厂模式

#### 3.代理模式

#### 4.观察者模式

#### 5.策略模式

## 二十五、全站搜索

## Java8 lambda表达式10个示例

<http://www.importnew.com/16436.html>

## 二十五、常用的Linux命令和Shell脚本

### 1.部署至 Linux + tomcat 服务器中常用的指令

**查看线程 ：**

ps -ef|grep tomcat名字（查看当前自己的项目是否在运行）

**杀掉线程 ：**

kill 进程id 杀进程（关闭当前运行的某个tomcat）

**启动tomcat ：**

./startup.sh （必须在bin下的才能启动）

**进入文件夹:**

cd 文件夹名称

**回到上一级 ：**

cd .. /

**查看控制台输出 :**

tail -f catalina.out （进入tomcat 中的 logs 文件夹里面）

**打开文件选择器将本地文件上传到服务器中 ：**

rz

**将服务器中的文件下载文件到本地 ：**

sz 文件名

**解压zip文件 ：**

unzip 文件名.zip

**将服务器中的文件夹，压缩成 tar.gz 格式文件：**

tar -zcvf 压缩文件名.tar.gz 被压缩文件名

**解压tar.gz 文件：**

tar -zxvf file.tar.gz

**删除文件/文件夹：**

rm -rf 文件夹 （-rf 没有回收站，一删除就找不回来了，删除的时候慎重）

**从普通用户切换到root用户 ：**

su root （当权限不足，需要root权限时，在SSH连接模式下,输入 su root ，回车，然后输入Linux 安装时输入的root密码，回车，密码正确即可切换为root账户）

## 二十六、Tomcat优化

## 二十七、MongoDB

## 二十八、消息中间件

### 1.RabbitMQ

2.

3.ActivityMQ

4.kafuka

## 二十九、开源项目

## 三十、Java 优化部分

### 1. Java程序员最常犯的错误！

**package** com.xgb.java.base.ArrayList;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.HashSet;

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.Set;

**import** org.apache.commons.lang3.ArrayUtils;

**public** **class** ListTest {

/\*\*

\* 一 、把数组转成ArrayList

\*/

**public** **static** **void** arrayToList() {

/\*\*

\* 使用Arrays.asList()方法可以得到一个ArrayList，但是得到这个ArrayList其实是定义在Arrays类中的一个私有的静态内部类。

\* 这个类虽然和java.util.ArrayList同名，但是并不是同一个类。java.util.Arrays.ArrayList类中实现了set(), get(),

\* contains()等方法，但是并没有定义向其中增加元素的方法。也就是说通过Arrays.asList()得到的ArrayList的大小是固定的。

\*/

String[] arr = {"0","1","2","3"};

List list = Arrays.*asList*(arr);

/\*\*

\* 如果在开发过程中，想得到一个真正的ArrayList对象（java.util.ArrayList的实例），可以通过以下方式：

\*/

ArrayList<String> arrayList = **new** ArrayList<String>(Arrays.*asList*(arr));

}

/\*\*

\* 二、判断一个数组是否包含某个值

\*/

**public** **static** **void** isContainsStr(String str) {

String[] arr = {"0","1","2","3"};

/\*\*

\* 虽然可以实现功能，但是效率却比较低。因为将数组压入Collection类型中，首先要将数组元素遍历一遍，然后再使用集合类做其他操作。

\*/

Set<String> set = **new** HashSet<String>(Arrays.*asList*(arr));

**boolean** flag = set.contains(str);

System.***out***.println("Is number flag: " + flag);

/\*\*

\* 在判断一个数组是否包含某个值的时候，推荐使用for循环遍历的形式或者使用Apache Commons类库中提供的ArrayUtils类的contains方法。

\*/

**boolean** flag2 = **false**;

**for**(**int** i = 0;i < arr.length;i ++) {

**if**(arr[i].contains(str)) {

flag2 = **true**;

}

}

System.***out***.println("Is number flag2: " + flag2);

/\*\*

\* Apache Commons类库中提供的ArrayUtils类的contains方法。

\*/

**boolean** flag3 = ArrayUtils.*contains*(arr, str);

System.***out***.println("Is number flag3: " + flag3);

}

/\*\*

\* 三、在循环中删除列表中的元素

\*/

**public** **static** **void** deleteEleInArrayByFor() {

String[] arr = {"0","1","2","3"};

/\*\*

\* 以上代码的目的是想遍历删除list中所有元素，但是结果却没有成功。原因是忽

\* 略了一个关键的问题：当一个元素被删除时，列表的大小缩小并且下标也会随之变化，

\* 所以当你想要在一个循环中用下标删除多个元素的时候，它并不会正常的生效。

\*/

List<String> list = **new** ArrayList<String>(Arrays.*asList*(arr));

**for**(**int** i=0;i<list.size();i++){

list.remove(i);

}

System.***out***.println(list);

/\*\*

\* 也有些人知道以上代码的问题就由于数组下标变换引起的。所以，他们想到使用增强for循环的形式：

\*/

List<String> list2 = **new** ArrayList<String>(Arrays.*asList*(arr));

**for**(String s:list2){

**if**(s.equals("a")){

list2.remove(s);

}

}

System.***out***.println(list2);

}

/\*\*

\* 四、HashTable 和 HashMap 的选择

\* 了解算法的人可能对HashTable比较熟悉，因为他是一个数据结构的名字。但在Java里边，用HashMap来表示这样的数据结构。

\* Hashtable和 HashMap的一个关键性的不同是，HashTable是同步的，而HashMap不是。所以通常不需要HashTable，HashMap用的更多。

\*/

**public** **static** **void** hashMapAndTable(){

}

/\*\*

\* 五、使用原始集合类型

\* 使用原始集合类型是很危险的，因为原始集合类型跳过了泛型类型检查，是不安全的。Set、Set< ? >和Set< Object >之间有很大差别。

\*/

/\*\*

\* 六、访问级别

\* 程序员们经常使用public作为类中的字段的修饰符，因为这样可以很简单的通过引用得到值，但这并不是好的设计，按照经验，分配给成员变量的访问级别应该尽可能的低。

\*/

/\*\*

\* 七、ArrayList与LinkedList的选择

\* 当程序员们不知道ArrayList与LinkedList的区别时，他们经常使用ArrayList，因为它看起来比较熟悉。然而，它们之前有巨大的性能差别。

\* 在ArrayList vs LinkedList vs Vector 区别、Java中常见亲属比较等文章中介绍过，简而言之，如果有大量的增加删除操作并且没有很多的随机访问元素的操作，

\* 应该首先LinkedList。（LinkedList更适合从中间插入或者删除（链表的特性））

\*/

**public** **static** **void** linkedListTest(){

LinkedList<String> linkedList = **new** LinkedList<String>();

linkedList.add("hello ");

linkedList.add("linkedList ");

linkedList.add("is ");

linkedList.add("List ");

System.***out***.println("LinkedList : " + linkedList.size());

}

**public** **static** **void** main(String[] args) {

*linkedListTest*();

}

}

## 三十一、亮点打造

1.数据读写分离

2.商品秒杀

3.高并发下分流处理