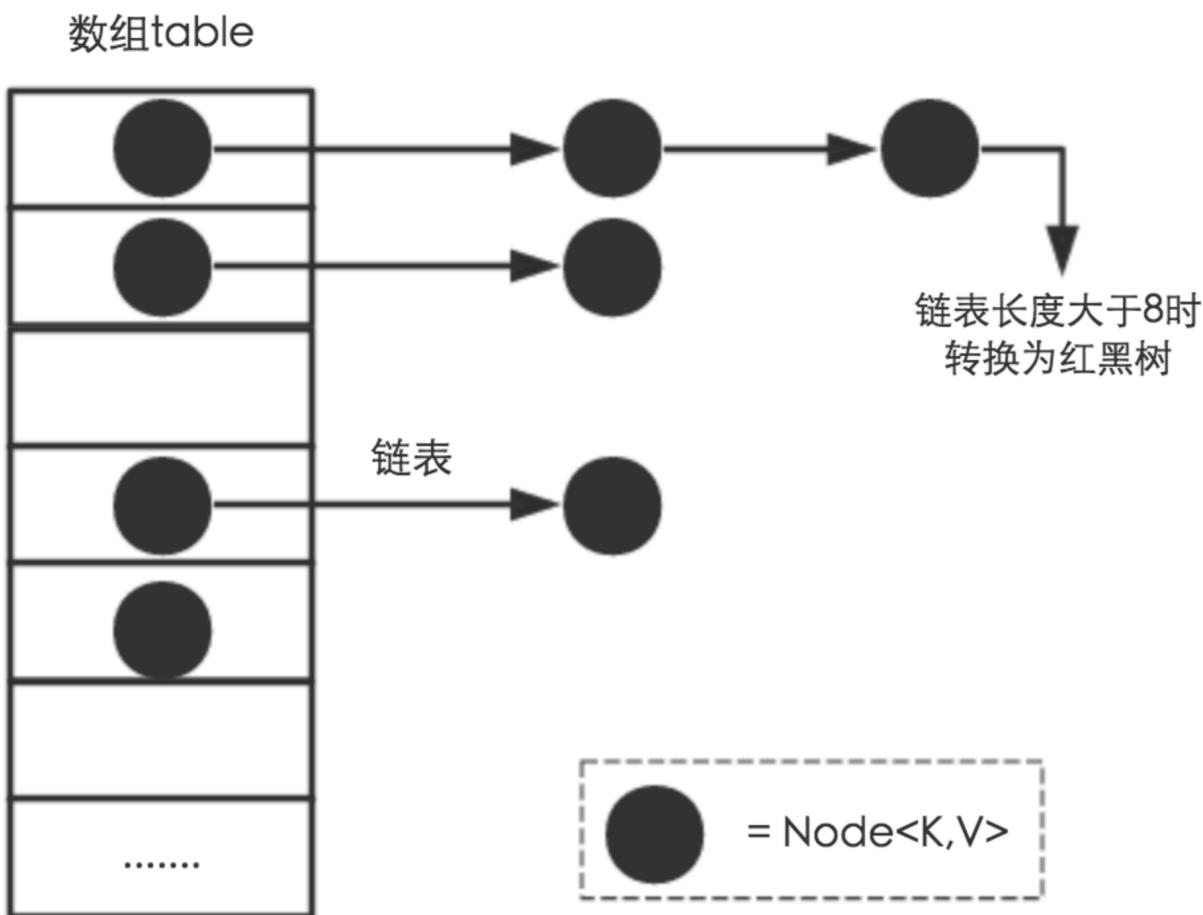


0x1 Java

001 语言基础

1. HashMap从入门到熟悉 ★★★★★★

1. hash碰撞的解决方案



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

2. JDK8中链表转红黑树，是否存在红黑树转链表

存在

3. 扩容发生的时间，为什么扩容是2倍，扩容的过程

1. 扩容发生的时间

大于等于阈值—即当前数组的长度乘以加载因子的值的时候，就要自动扩容。

- 负载因子

- 过小：容易发生reszie，消耗性能
- 过大：容易发生hash碰撞，链表变长，红黑树变高

2. 为什么 hashmap 底层数组要保证是2的n次方

```
//hash值的计算分为两步：
//1. 异或运算
static final int hash(Object key) {    //jdk1.8 & jdk1.7
    int h;
    // h = key.hashCode() 为第一步 取hashCode值
    // h ^ (h >>> 16) 为第二步 高位参与运算
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

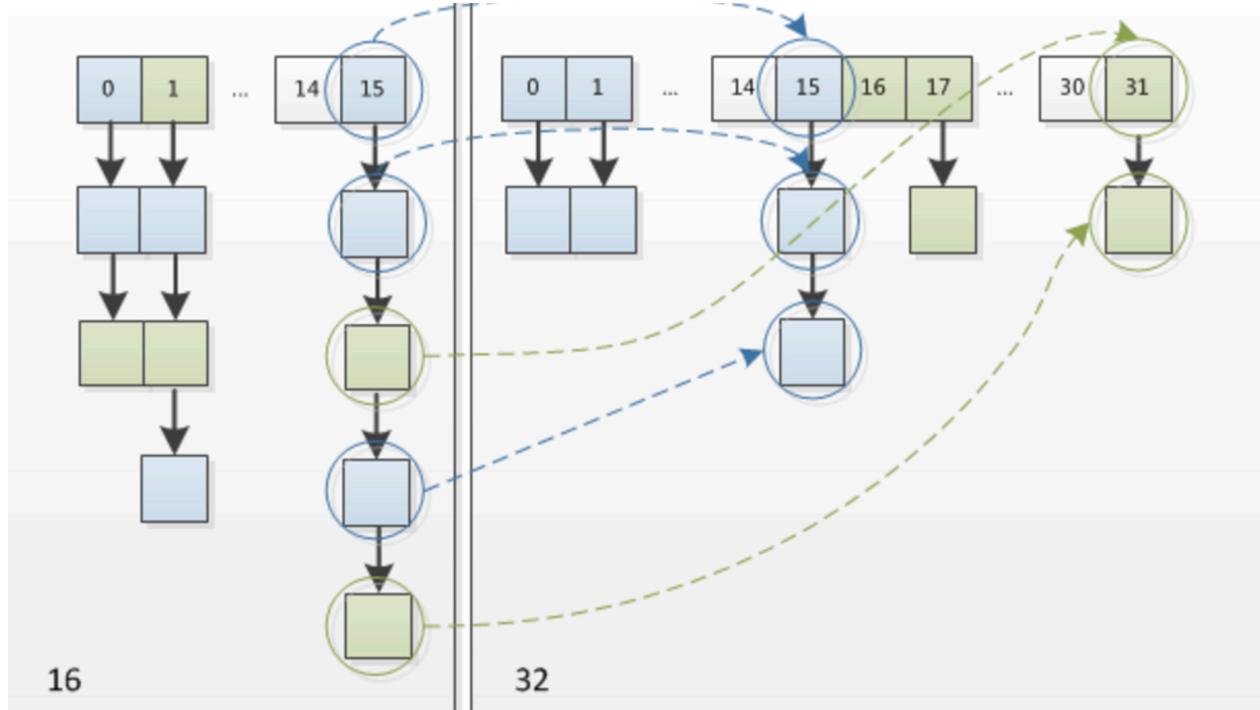
//2. 和数组长度与运算，分布到原有数组中
hash = h & (n-1)
```

得到 hash 值之后，再与数组的长度-1 (length-1) 进行一次与运算，因为如果数组的长度是 2 的倍数，那么length-1 的二进制一定是 ...00001111...这种形式，也就是前面一定都是 0，后面全是1，那么再与 hash 值进行与运算的时候，结果一定是在原来数组大小的范围内，比如默认数组大小 $16-1=15$ 的二进制为：**00000000 00000000 00000000 00001111**，某 key 的hash 值为：11010010 00000001 10010000 00100100，那么与上面做与运算的时候，值会对后面的四位进行运算，肯定会落在0~15 的范围内，假如不是 2 的倍数，那么 length-1 的二进制后面就不可能全是 1，做与运算的时候就会造成空间浪费。

3. 扩容的具体过程

- 开辟了新的数组空间
- 元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

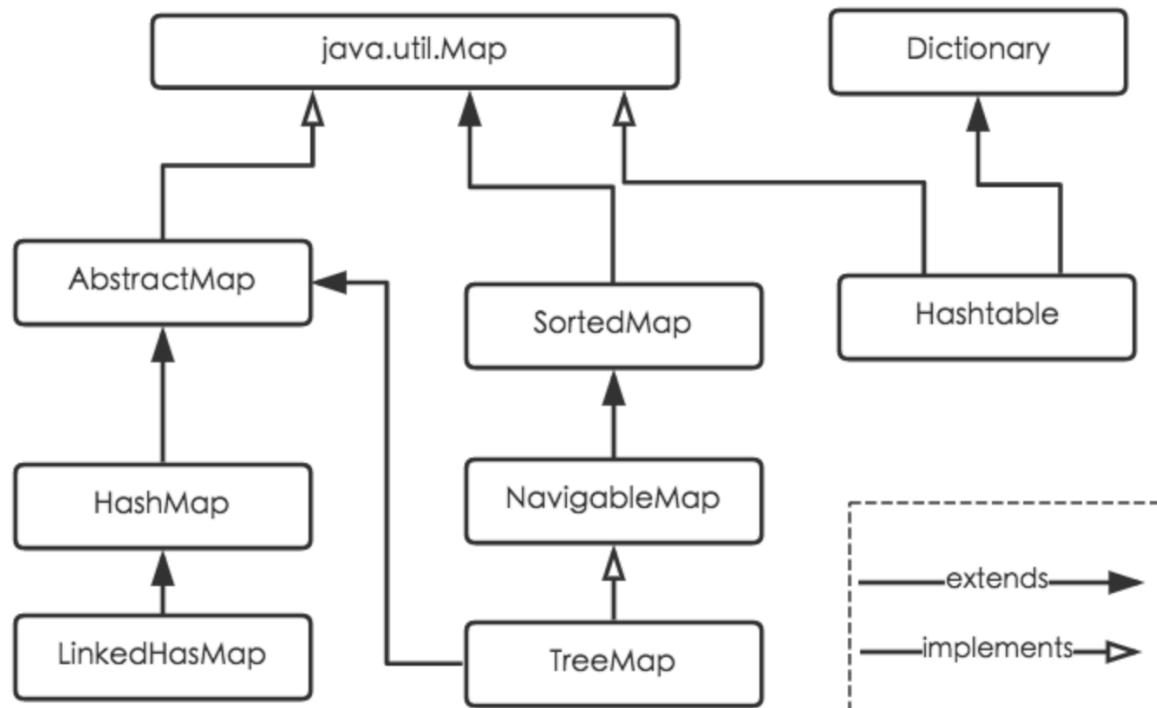
我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



4. 既然存在扩容，是否存在缩容

没有缩容机制，没有看到与resize()对应方法。

5. HashMap和HashTable、HashSet、LinkedHashMap



- **Hashtable:** **Hashtable**是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中

使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

- LinkedHashMap： LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。
- TreeMap： TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

1. HashMap和HashSet的区别

HashMap	HashSet
HashMap实现了Map接口	HashSet实现了Set接口
HashMap储存键值对	HashSet仅仅存储对象
使用put()方法将元素放入map中	使用add()方法将元素放入set中
HashMap中使用键对象来计算hashcode值	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false
HashMap比较快，因为是使用唯一的键来获取对象	HashSet较HashMap来说比较慢

6. Hashmap为什么是线程不安全的

- 表面原因
 - Hashmap的方法没有使用synchronized进行同步
- 实际原因
 - 如果能找到并发环境下的问题，就能证明是不安全的
 - 并发环境下， hashmap进入扩容的时候容易造成Entry链成环，在查询等操作的时候容易造成死循环

7. TreeMap和HashMap有什么区别

使用Iterator迭代器遍历的时候，HashMap的结果是没有排序的，而TreeMap输出的结果是排好序的。

Ref2: <https://yikun.github.io/2015/04/01/Java-HashMap%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86%E5%8F%8A%E5%AE%9E%E7%8E%BO/>

2. ConcurrentHashMap

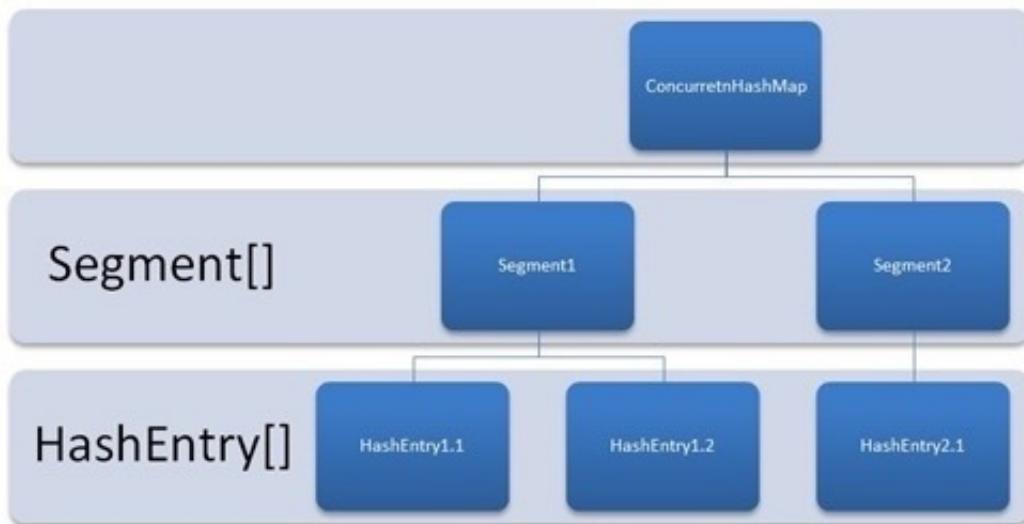
1. ConcurrentHashMap是如何实现线程安全的

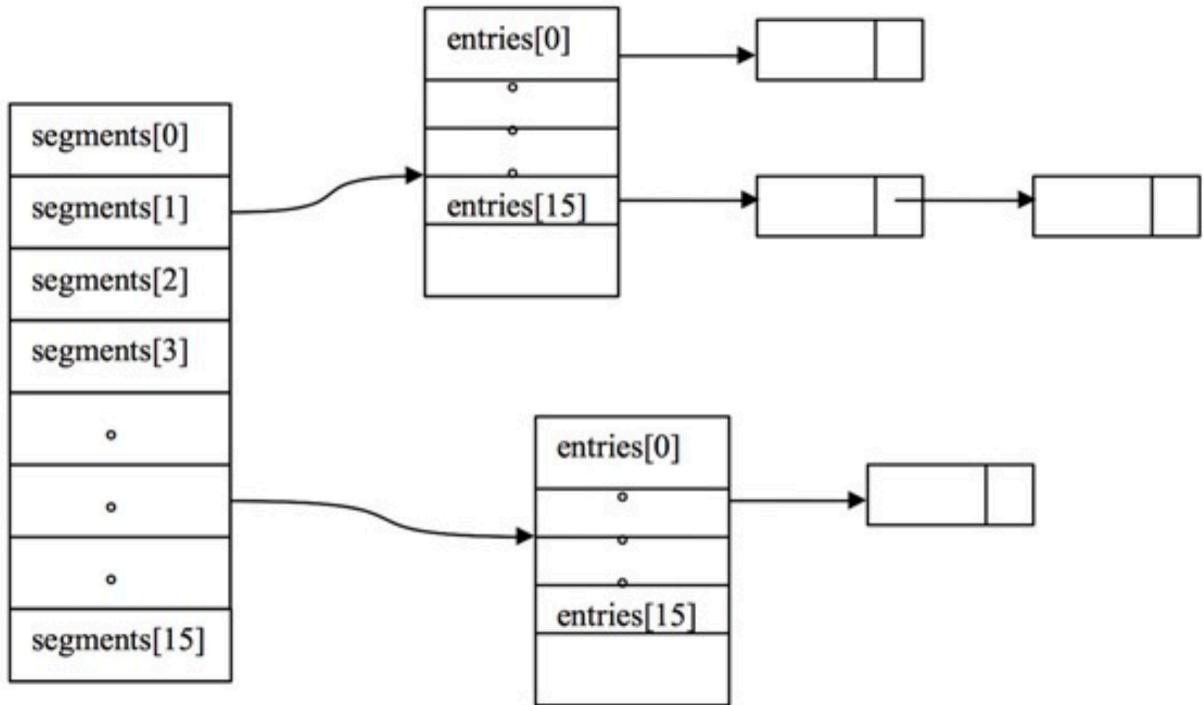
1. 数据结构

```
static final class Segment<K,V> extends ReentrantLock implements Serializable
{
    private static final long serialVersionUID = 2249069246763182397L;

    // 和 HashMap 中的 HashEntry 作用一样，真正存放数据的桶
    transient volatile HashEntry<K,V>[] table;
    transient int count;
    transient int modCount;
    transient int threshold;
    final float loadFactor;

}
```

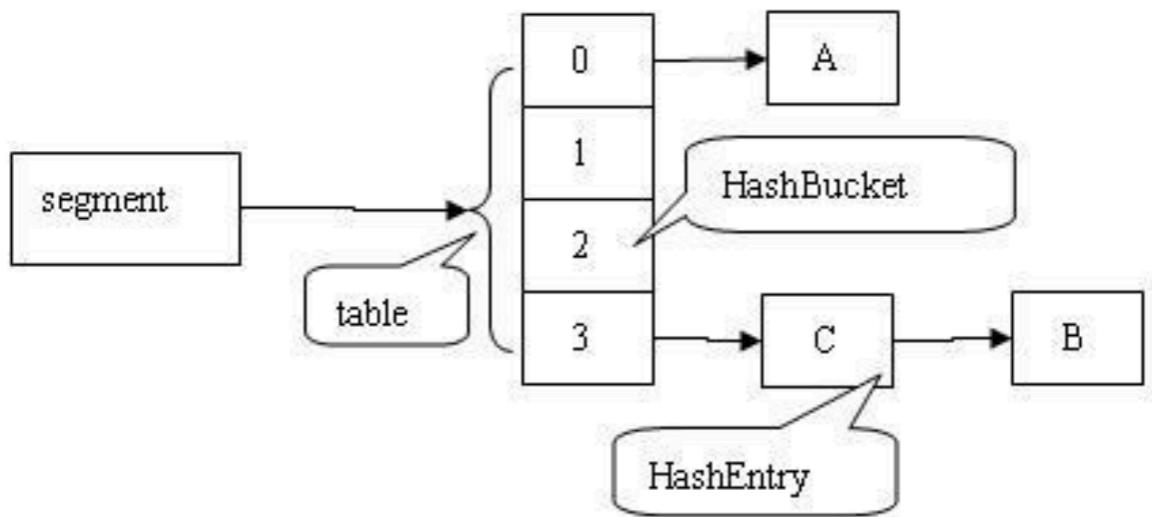




Segment 类继承于 **ReentrantLock** 类，从而使得 **Segment** 对象能充当锁的角色。

本质上，**ConcurrentHashMap**就是一个**Segment**数组，而一个**Segment**实例则是一个小的哈希表。由于**Segment**类继承于**ReentrantLock**类，从而使得**Segment**对象能充当锁的角色，这样，每个**Segment**对象就可以守护整个**ConcurrentHashMap**的若干个桶，其中每个桶是由若干个**HashEntry**对象链接起来的链表。

2. 读操作的并发性



上面的HashBucket就是entries数组

HashEntry用来封装具体的键值对，是个典型的四元组。与**HashMap**中的**Entry**类似，**HashEntry**也包括同样的四个域，分别是key、hash、value和next。不同的是，在**HashEntry**类中，key, hash和next域都被声明为final的，value域被volatile所修饰，因此**HashEntry**对象几乎是不可变的，这是**ConcurrentHashMap**读操作并不需要加锁的一个重要原因。**next**域被声明为final本身就意味着我们不能从**hash**链的中间或尾部添加或删除节点，因为这需要修改**next**引用值，因此所有的节点的修改只能

从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制(重新new)一遍，最后一个节点指向要删除结点的下一个结点(这在谈到ConcurrentHashMap的删除操作时还会详述)。特别地，由于value域被volatile修饰，所以其可以确保被读线程读到最新的值，这是ConcurrentHashMap读操作并不需要加锁的另一个重要原因。

3. 写操作的并发性

segmentFor()方法根据传入的hash值向右无符号右移segmentShift位，然后和segmentMask进行与操作就可以定位到特定的段。在这里，假设Segment的数量(segments数组的长度)是2的n次方(Segment的数量总是2的倍数，具体见构造函数的实现)，那么segmentShift的值就是32-n(hash值的位数是32)，而segmentMask的值就是 2^{n-1} (写成二进制的形式就是n个1)。进一步地，我们就可以得出以下结论：根据key的hash值的高n位就可以确定元素到底在哪一个Segment中。

```
final Segment<K,V> segmentFor(int hash) {
    return segments[(hash >> segmentShift) & segmentMask];
}
```

2. ConcurrentHashMap的性能如何

- 针对读写操作进入分析
- 在ConcurrentHashMap中，无论是读操作还是写操作都能保证很高的性能：在进行读操作时(几乎)不需要加锁，而在写操作时通过锁分段技术只对所操作的段加锁而不影响客户端对其它段的访问。特别地，在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作 (如果并发级别设为16)，及任意数量线程的读操作。

3. JDK1.7到JDK1.8发生的变化

参考：<https://blog.csdn.net/bolang789/article/details/79855053>

3. 说一说你对java.lang.Object对象中的hashCode和equals方法的理解，在什么场景下需要重新实现这两个方法★★★★

3.1 equals方法的作用

equals() 方法是用来判断其他的对象是否和该对象相等，equals()方法在object类中定义如下：

```
public boolean equals(Object obj) {
    return (this == obj); //比较的是地址
}
```

Override Equals方法时的规则：

- **自反性**：对于任何非空引用值 x，x.equals(x) 都应返回 true。
- **对称性**：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

- **传递性**: 对于任何非空引用值 x、y 和 z, 如果 x.equals(y) 返回 true, 并且 y.equals(z) 返回 true, 那么 x.equals(z) 应返回 true。
- **一致性**: 对于任何非空引用值 x 和 y, 多次调用 x.equals(y) 始终返回 true 或始终返回 false, 前提是对象上 equals 比较中所用的信息没有被修改。
- **非空性**: 对于任何非空引用值 x, x.equals(null) 都应返回 false。

3.2 重写equals为什么要重写hashCode

为了保证使用Map接口时, “相同”对象的hashCode也是相同的

HashMap对象是根据其Key的hashCode来获取对应的Value。

在重写父类的equals方法时, 也重写hashCode方法, 使相等的两个对象获取的HashCode也相等, 这样当此对象做Map类中的Key时, 两个equals为true的对象其获取的value都是同一个, 比较符合实际。

3.3 equals和hashCode的关系

一个好的hashCode的方法的目标: 为不相等的对象产生不相等的散列码, 同样的, 相等的对象必须拥有相等的散列码。

如果两个对象equals, 那么它们的hashCode必然相等, 但是hashCode相等, equals不一定相等。

4. 

如何深入研究Java里的课题: <https://blog.csdn.net/javazejian/article/details/73413292>

5. Java中的几种基本数据类型是什么, 各占用多少字节

以int 32位作为基准, 其他的double or half

序号	数据类型	位数	默认值	取值范围	举例说明
1	byte(位)	8	0	-2 ⁷ - 2 ⁷ -1	byte b = 10;
2	short(短整数)	16	0	-2 ¹⁵ - 2 ¹⁵ -1	short s = 10;
3	int(整数)	32	0	-2 ³¹ - 2 ³¹ -1	int i = 10;
4	long(长整数)	64	0	-2 ⁶³ - 2 ⁶³ -1	long l = 10l;
5	float(单精度)	32	0.0	-2 ³¹ - 2 ³¹ -1	float f = 10.0f;
6	double(双精度)	64	0.0	-2 ⁶³ - 2 ⁶³ -1	double d = 10.0d;
7	char(字符)	16	空	0 - 2 ¹⁶ -1	char c = 'c';
8	boolean(布尔值)	8	false	true, false	boolean b = true;

6. String类能够被继承吗，为什么

- 直接原因：不能，被final修饰。
- 根本原因：（1）为了安全考虑，比如下面的例子，在将String参数传入函数之后不会发生改变，对比StringBuffer （2）为了效率考虑：在大量使用字符串的时候，指向的是常量池中同一个常量，节省内存空间，提高效率

```
class Test{
    //不可变的String
    public static String appendStr(String s){
        s+="bbb";
        return s;
    }

    //可变的StringBuilder
    public static StringBuilder appendSb(StringBuilder sb){
        return sb.append("bbb");
    }

    public static void main(String[] args){
        //String做参数
        String s=new String("aaa");
        String ns=Test.appendStr(s);
        System.out.println("String aaa >> "+s.toString());

        //StringBuilder做参数
        StringBuilder sb=new StringBuilder("aaa");
        StringBuilder nsb=Test.appendSb(sb);
        System.out.println("StringBuilder aaa >> "+sb.toString());
    }
}

//Output:
//String aaa >> aaa
//StringBuilder aaa >> aaabbb
```

7. 反射机制 ★★★★★

1. 实现方式

反射就是动态加载对象，并对对象进行剖析。在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法，这种动态获取信息以及动态调用对象方法的功能成为Java反射机制。

获取Class对象三种方式：

- 通过对象.getClass () 方式
- 通过类名.Class 方式
- 通过Class.forName 方式

```

/**
 * 通过反射获取类的所有变量
 */
private static void printFields(){
    //主要是首先获取对象的class类
    //1.获取并输出类的名称
    Class mClass = SonClass.class;
    System.out.println("类的名称: " + mClass.getName());

    //2.1 获取所有 public 访问权限的变量
    // 包括本类声明的和从父类继承的
    Field[] fields = mClass.getFields();

    //2.2 获取所有本类声明的变量 (不问访问权限)
    //Field[] fields = mClass.getDeclaredFields();

    //3. 遍历变量并输出变量信息
    for (Field field :
        fields) {
        //获取访问权限并输出
        int modifiers = field.getModifiers();
        System.out.print(Modifier.toString(modifiers) + " ");
        //输出变量的类型及变量名
        System.out.println(field.getType().getName()
            + " " + field.getName());
    }
}

```

2. 性能分析

1. 优点

增加程序的灵活性，避免将程序写死到代码里。

2. 缺点

性能问题

1. 使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此Java反射机制主要应用在对灵活性和扩展性要求很高的系统框架上，普通程序不建议使用。

2. 反射包括了一些动态类型，所以JVM无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。

模糊程序内部逻辑

程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂。

安全限制

使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如Applet，那么这就是个问题了。

8. String, StringBuffer, StringBuilder

1. String 和 StringBuffer/StringBuilder

1、String类型和StringBuffer类型的主要性能区别：String是不可变的对象，因此每次在对String类进行改变的时候都会生成一个新的string对象，然后将指针指向新的string对象，所以经常要改变字符串长度的话不要使用string，因为每次生成对象都会对系统性能产生影响，特别是当内存中引用的对象多了以后，JVM的GC就会开始工作，性能就会降低；

2、使用StringBuffer类时，每次都会对StringBuffer对象本身进行操作，而不是生成新的对象并改变对象引用，所以多数情况下推荐使用StringBuffer，特别是字符串对象经常要改变的情况；

2. StringBuffer和StringBuilder

StringBuffer类是线程安全的，StringBuilder不是线程安全的；

9. “假泛型”

1. 泛型为什么不安全

泛型能够保证容器里存放元素的类型，但是通过运行时的反射机制能够绕过泛型的编译验证。

```
public static void main(String[] args) throws Exception {
    List<Integer> list = new ArrayList<>();
    list.add(1);
    //list.add("a"); // 这样直接添加肯定是不允许的

    //下面通过java的反射，绕过泛型来给添加字符串
    Method add = list.getClass().getMethod("add", Object.class);
    add.invoke(list, "a");

    System.out.println(list); // [1, a] 输出没有没问题
    System.out.println(list.get(1)); // a
}
```

2. 泛型擦除

Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉。这个过程就称为类型擦除。

如在代码中定义的List和List等类型，在编译后都会编程List。JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。类型擦除也是Java的泛型实现方法与C++模版机制实现方式之间的重要区别。

10. 深拷贝和浅拷贝的区别 ★★★★

创建对象的五种方式，**Object类的clone方法是重点：**

①、通过 new 关键字

这是最常用的一种方式，通过 new 关键字调用类的有参或无参构造方法来创建对象。比如 Object
obj = new Object();

②、通过 Class 类的 newInstance() 方法

这种默认是调用类的无参构造方法创建对象。比如 Person p2 = (Person)
Class.forName("com.ys.test.Person").newInstance();

③、通过 Constructor 类的 newInstance 方法

这和第二种方法类似，都是通过反射来实现。通过 java.lang.reflect.Constructor 类的
newInstance() 方法指定某个构造器来创建对象。

```
Person p3 = (Person) Person.class.getConstructors()[0].newInstance();
```

实际上第二种方法利用 Class 的 newInstance() 方法创建对象，其内部调用还是 Constructor 的
newInstance() 方法。

④、利用 Clone 方法

Clone 是 Object 类中的一个方法，通过 对象A.clone() 方法会创建一个内容和对象 A 一模一样的对
象 B，clone 克隆，顾名思义就是创建一个一模一样的对象出来。

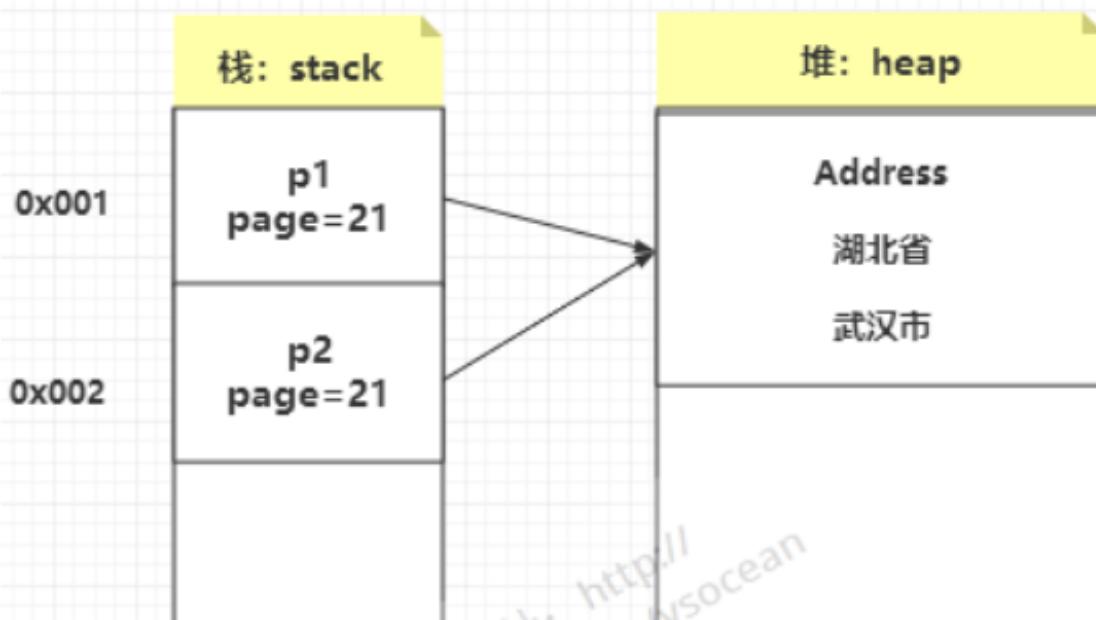
```
Person p4 = (Person) p3.clone();
```

⑤、反序列化

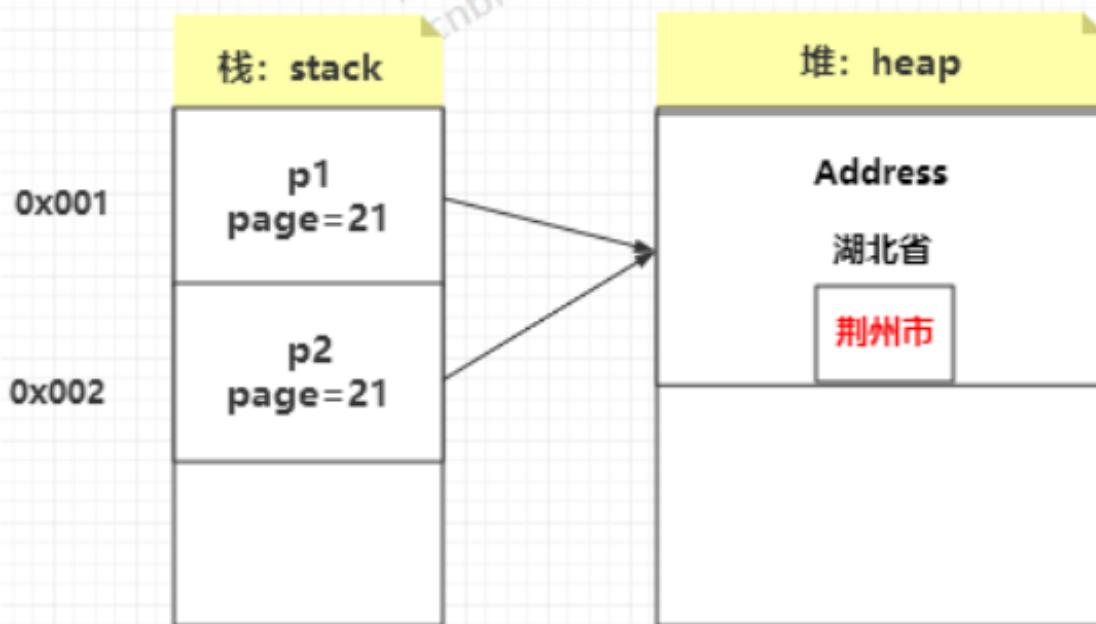
序列化是把堆内存中的 Java 对象数据，通过某种方式把对象存储到磁盘文件中或者传递给其他网
络节点（在网络上传输）。而反序列化则是把磁盘文件中的对象数据或者把网络节点上的对象数据，恢
复成Java对象模型的过程。

浅拷贝通过**Object.clone()**方法实现：

浅拷贝：引用类型只复制引用

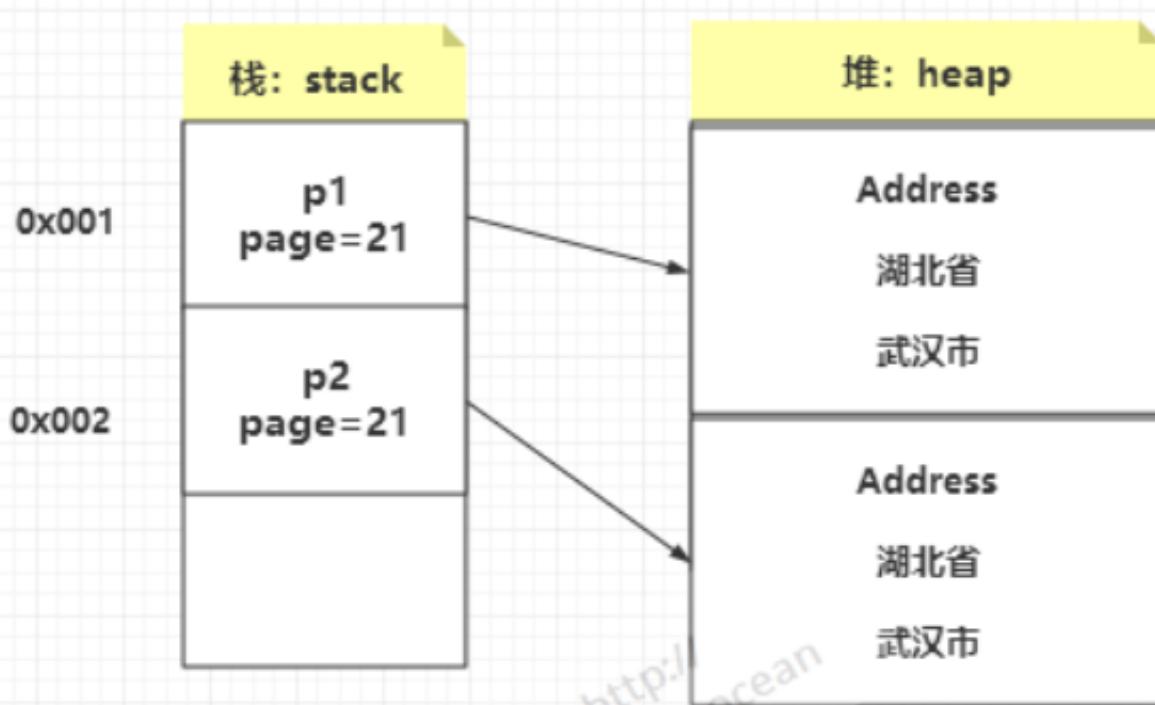


将 p2 的 **Address** 修改为 湖北省，荆州市

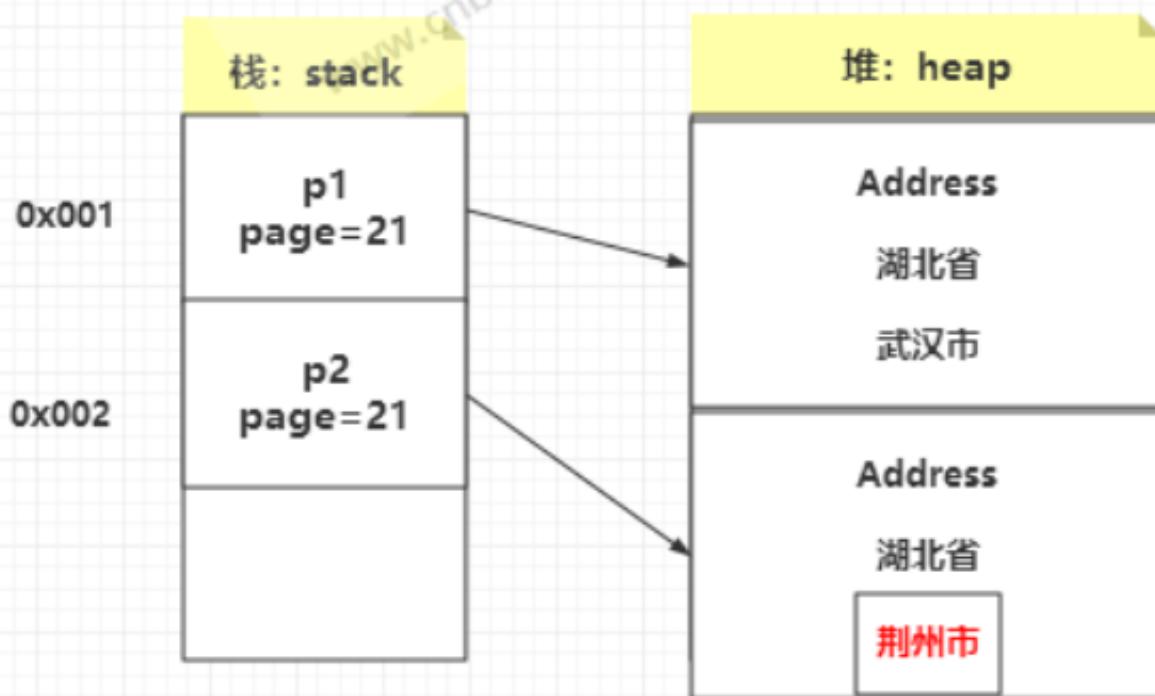


浅拷贝：创建一个新对象，然后将当前对象的非静态字段复制到该新对象，如果字段是值类型的，那么对该字段执行复制；如果该字段是引用类型的话，则复制引用但不复制引用的对象。因此，原始对象及其副本引用同一个对象。

深拷贝：所有属性都复制独立一份



将 p2 的 Address 修改为 湖北省, 荆州市



深拷贝：创建一个新对象，然后将当前对象的非静态字段复制到该新对象，无论该字段是值类型的还是引用类型，都复制独立的一份。当你修改其中一个对象的任何内容时，都不会影响另一个对象的内容。

请问如何实现对象的深拷贝？

序列化是将对象写到流中便于传输，而反序列化则是把对象从流中读取出来。这里写到流中的对象则是原始对象的一个拷贝，因为原始对象还存在 JVM 中，所以我们可以利用对象的序列化产生克隆对象，然后通过反序列化获取这个对象。

注意每个需要序列化的类都要实现 Serializable 接口，如果有某个属性不需要序列化，可以将其声明为 transient，即将其排除在克隆属性之外。

```
//深度拷贝
public Object deepClone() throws Exception{
    // 序列化
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(bos);

    oos.writeObject(this);

    // 反序列化
    ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
    ObjectInputStream ois = new ObjectInputStream(bis);

    return ois.readObject();
}
```

11. ArrayList和LinkedList的区别，有线程安全的类吗? ★★★

- ArrayList是一个可以处理变长数组的类型，这里不局限于“数组”，ArrayList是一个泛型类，可以存放任意类型的对象。
- LinkedList可以看做为一个双向链表，所有的操作都可以认为是一个双向链表的操作，因为它实现了Deque接口和List接口。
- Vector也是一个类似于ArrayList的可变长度的数组类型，它的内部也是使用数组来存放数据对象的。值得注意的是Vector与ArrayList唯一的区别是，Vector是线程安全的，即它的大部分方法都包含有关键字synchronized。若对于单一线程的应用来说，最好使用ArrayList代替Vector。

13. 面向对象的三大特征

1. 封装

封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过外部接口，以特定的访问权限来使用类的成员。

1. public private protected default关键词的区别

	同一个类	同一个包	不同包的子类	不同包
Private	√			
Default	√	√		
Protected	√	√	√	
Public	√	√	√	√

2. 继承

继承就是子类继承父类的特征和行为

- 抽象类和接口的区别，类可以继承多个类吗，接口可以继承多个接口吗，类可以实现多个接口吗

3. 多态

引用可以指向父类对象或者子类对象

- Override和Overload的区别

- Override：子类覆盖父类的方法
- Overload：同样的方法名，不同的参数以及参数类型，可以被当做不同的函数

14. 线程池-从入门到熟悉 ★★★★★

- 为什么需要线程池
 - 由于创建和销毁线程都需要很大的开销，运用线程池就可以大大的缓解这些内存开销很大的问题。
- 为什么需要设置线程池参数
 - 整线程池中工作线程的数目，防止因为消耗过多的内存。
 - CPU密集型尽量少线程数；IO密集型尽量多线程数，压榨CPU性能

1. newSingleThreadExecutor

```
public static void main(String[] args) {
    ExecutorService pool = Executors.newSingleThreadExecutor();
    for (int i = 0; i < 10; i++) {
        pool.execute(() -> {
            System.out.println(Thread.currentThread().getName() + "\t开始发
车啦....");
        });
    }
}
```

- 这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。
 - 保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
 - 应用场景： **GUI单线程队列**

2. newFixedThreadPool

```
public class newFixedThreadPool {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(6); //总共有6个线程
        for (int i = 0; i < 10; i++) {
            pool.execute(() -> {
                System.out.println(Thread.currentThread().getName() + "\t开始发
车啦....");
            });
        }
    }
}
```

① 线程数少于核心线程数，也就是设置的线程数时，新建线程执行任务 ② 线程数等于核心线程数后，将任务加入阻塞队列，于队列容量非常大，可以一直加加加 ③ 执行完任务的线程反复去队列中取任务执行

- FixedThreadPool 用于负载比较重的服务器，为了资源的合理利用，需要限制当前线程数量。

3. newCachedThreadPool

```
public static void main(String[] args) throws InterruptedException {
    ExecutorService service = Executors.newCachedThreadPool();
    System.out.println(service);

    for (int i = 0; i < 20; i++) {
        service.execute(() -> {
            System.out.println(Thread.currentThread().getName());
        });
    }
    System.out.println(service);
    service.shutdown();
}
```

① 没有核心线程，直接向 SynchronousQueue 中提交任务 ② 如果有空闲线程，就去取出任务执行；如果没有空闲线程，就新建一个 ③ 执行完任务的线程有 60 秒生存时间，如果在这个时间内可以接到新任务，就可以继续活下去，否则会被销毁

4. newScheduledThreadPool

```
public static void main(String[] args) {
    ScheduledExecutorService pool = Executors.newScheduledThreadPool(10);
    for (int i = 0; i < 2; i++) {
        //每隔一秒执行一次
        pool.scheduleAtFixedRate(() -> {
            System.out.println(Thread.currentThread().getName() + "\t开始发
车啦....");
        }, 1, 1, TimeUnit.SECONDS);
    }
}
```

- 创建一个定长线程池，支持定时和周期性任务执行。

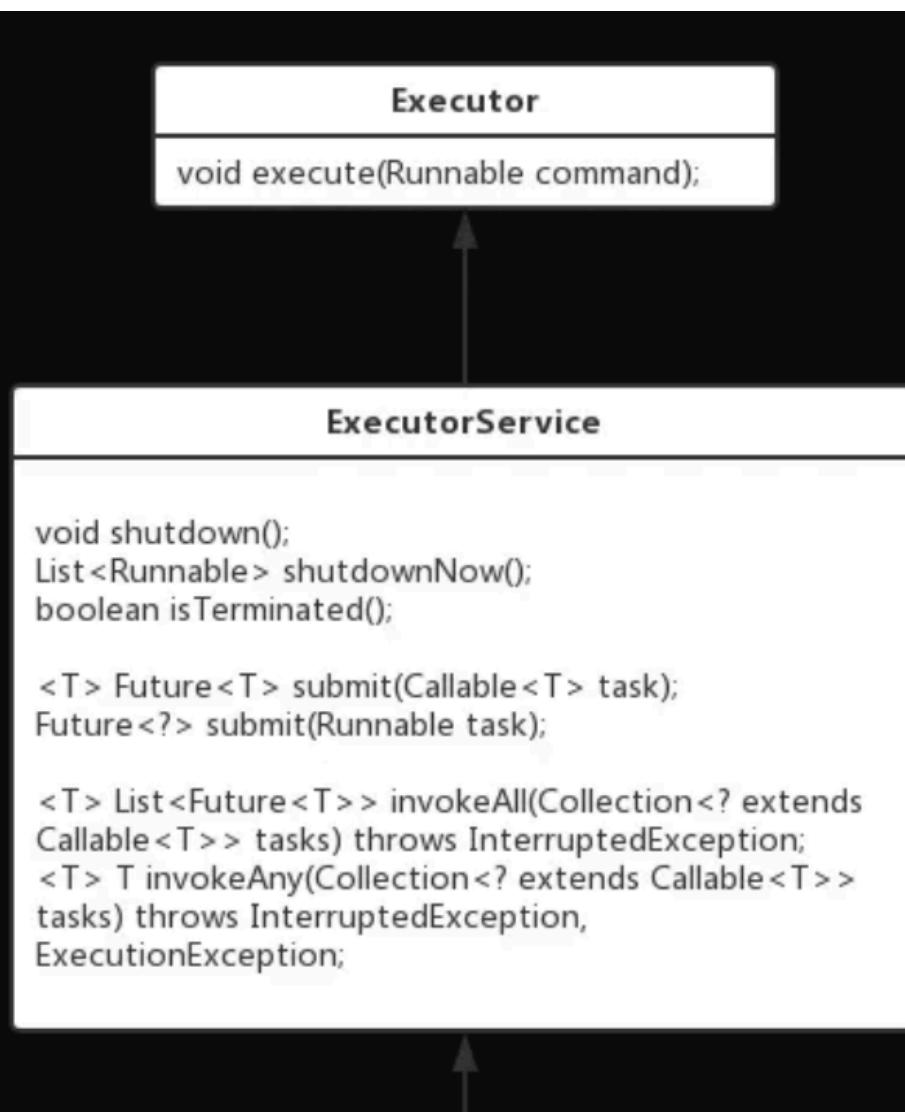
5. 关闭线程池方法

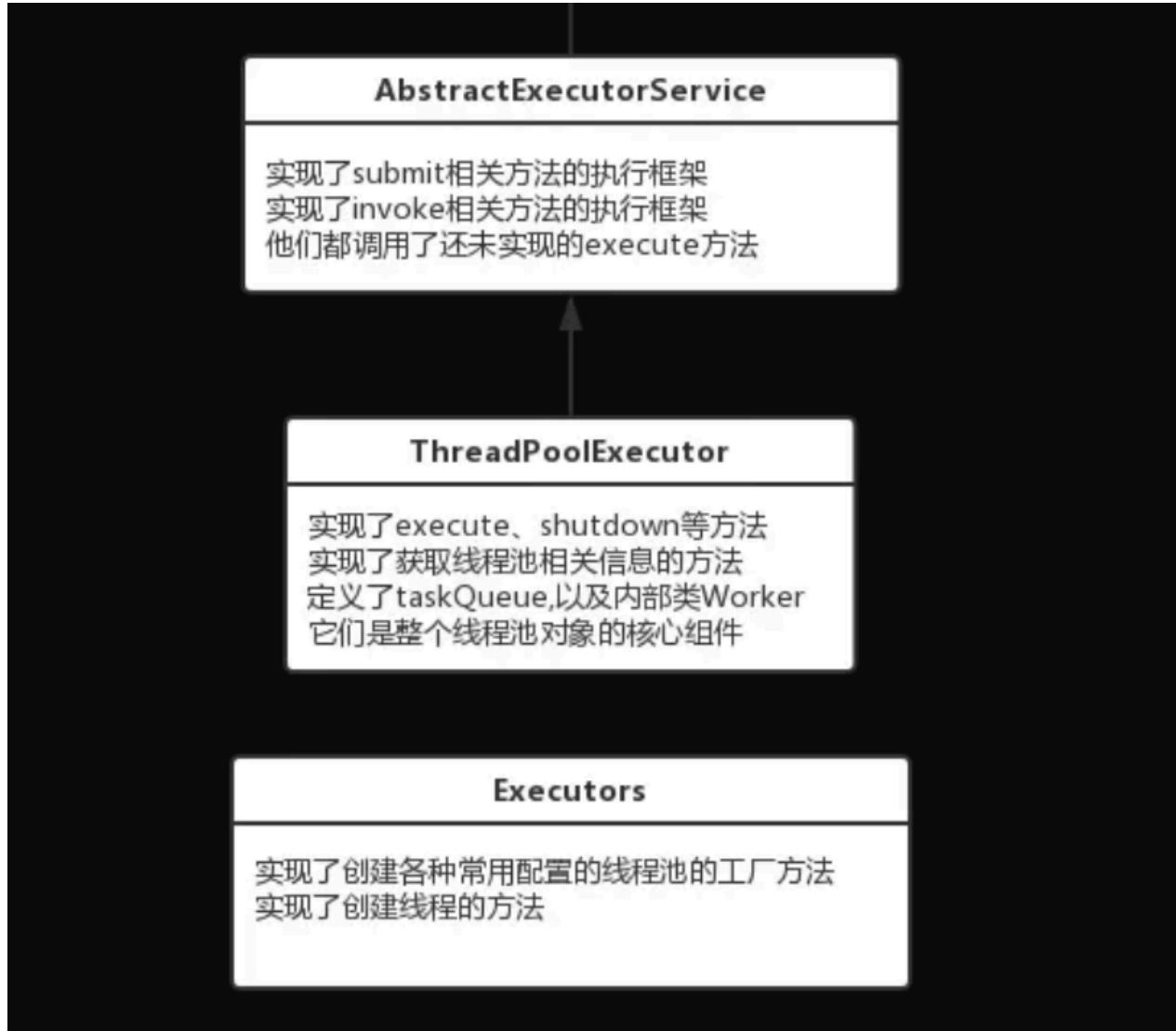
execute(): 提交不需要返回值的任务

submit(): 提交需要返回值的任务

pool.shutdown()

6. 线程池相关的类



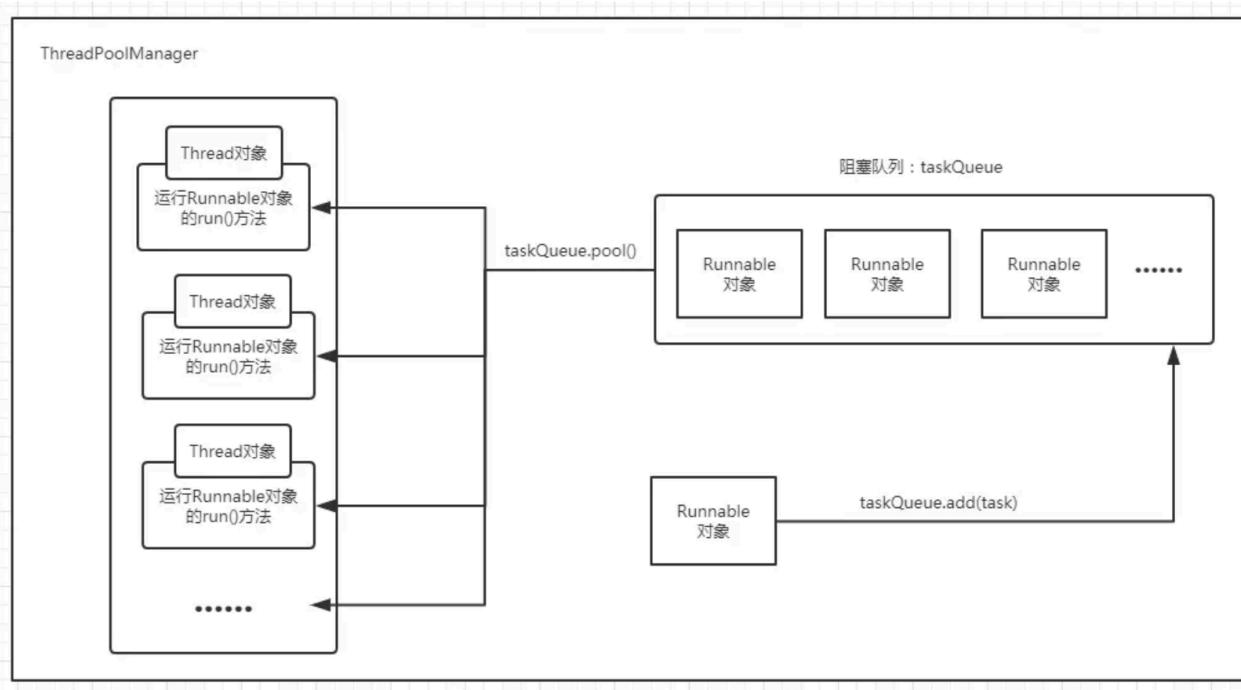


核心类主要是ThreadPoolExecutor和Executors工厂类

1. ThreadPoolExecutor

ThreadPoolExecutor是线程池的真正实现，他通过构造方法的一系列参数，来构成不同配置的线程池。

7. 线程池参数



Executors工厂类建立线程池是通过ThreadPoolExecutor设置不同参数建立。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    //忽略赋值与校验逻辑
}
```

- corePoolSize：默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中。
- maxPoolSize：当线程数大于或等于核心线程，且任务队列已满时，线程池会创建新的线程，直到线程数量达到maxPoolSize。如果线程数已等于maxPoolSize，且任务队列已满，则已超出线程池的处理能力，线程池会拒绝处理任务而抛出异常。
- keepAliveTime：当线程空闲时间达到keepAliveTime，该线程会退出，直到线程数量等于corePoolSize。如果allowCoreThreadTimeout设置为true，则所有线程均会退出直到线程数量为0。
- handler：任务拒绝策略，当运行线程数已达到maximumPoolSize，队列也已经装满时会调用该参数拒绝任务，有默认实现。

8. 什么是IO密集型，什么是CPU密集型

1. CPU密集

CPU密集型也叫**计算密集型**，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，**CPU Loading很高**。

在多重程序系统中，大部份时间用来做计算、逻辑判断等CPU动作的程序称之为CPU bound。例如一个计算圆周率至小数点一千位以下的程序，在执行的过程当中绝大部分时间用在三角函数和开根号的计算，便是属于CPU bound的程序。

CPU bound的程序一般而言CPU占用率相当高。这可能是因为任务本身不太需要访问I/O设备，也可能是因为程序是多线程实现因此屏蔽掉了等待I/O的时间。

2. IO密集

IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高。

I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力。

3. 针对IO密集型/CPU密集型，选择怎样的线程池参数

- 如果任务是IO密集型，一般线程数需要设置2倍CPU数以上，以此来尽量利用CPU资源。
- 如果任务是CPU密集型，一般线程数量只需要设置CPU数加1即可，更多的线程数也只能增加上下文切换，不能增加CPU利用率。

9.为什么要设置核心线程数和最大线程数

当提交一个新任务到线程池时 首先线程池判断基本线程池(corePoolSize)是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程； 其次线程池判断工作队列(workQueue)是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程； 最后线程池判断整个线程池(maximumPoolSize)是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务； 如果线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过 keepAliveTime，线程将被终止，直至线程池中的线程数目不大于 corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过 keepAliveTime，线程也会被终止。

这么比喻吧，核心线程数就像是工厂正式工，最大线程数，就是工厂临时工作量加大，请了一批临时工，临时工加正式工的和就是最大线程数，等这批任务结束后，临时工要辞退的，而正式工会留下。

10. 线程池工作顺序

```
If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.  
If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.  
If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.
```

corePoolSize -> 工作队列 -> maxPoolSize -> 拒绝策略

11. 线程池的拒绝策略

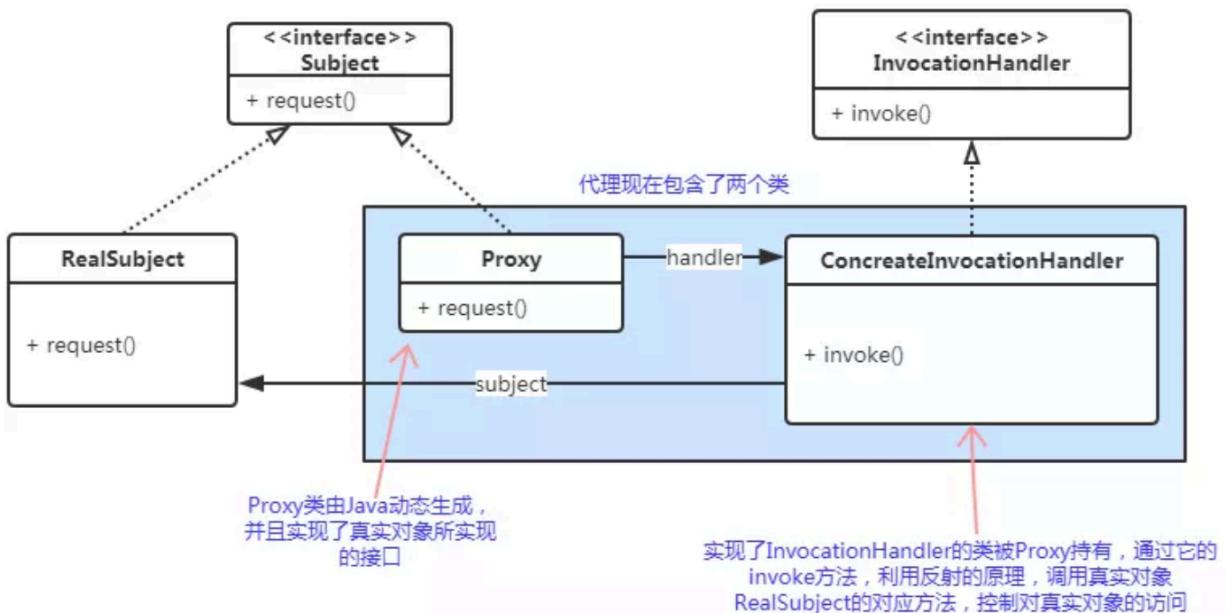
RejectedExecutionHandler类型

RejectedExecutionHandler	特性及效果
AbortPolicy	线程池默认的策略，如果元素添加到线程池失败，会抛出RejectedExecutionException异常
DiscardPolicy	如果添加失败，则放弃，并且不会抛出任何异常
DiscardOldestPolicy	如果添加到线程池失败，会将队列中最早添加的元素移除，再尝试添加，如果失败则按该策略不断重试
CallerRunsPolicy	如果添加失败，那么主线程会自己调用执行器中的execute方法来执行改任务
自定义	如果觉得以上几种策略都不合适，那么可以自定义符合场景的拒绝策略。需要实现RejectedExecutionHandler接口，并将自己的逻辑写在rejectedExecution方法内。 https://www.jianshu.com/p/3caa0c23a157

14. 动态代理

动态代理的两种方式：<https://www.jianshu.com/p/3caa0c23a157>

1. JDK动态代理



- (1) 创建被代理对象的接口类。
- (2) 创建具体被代理对象接口的实现类。
- (3) 创建一个InvocationHandler的实现类，并持有被代理对象的引用。然后在invoke方法中利用反射调用被代理对象的方法。
- (4) 利用Proxy.newProxyInstance方法创建代理对象，利用代理对象实现真实对象方法的调用。

1. 上层接口和实现类

```
public interface Subject {  
    void request();  
}  
  
public class RealSubject implements Subject {  
    @Override  
    public void request() {  
        System.out.println("request invoke");  
    }  
}
```

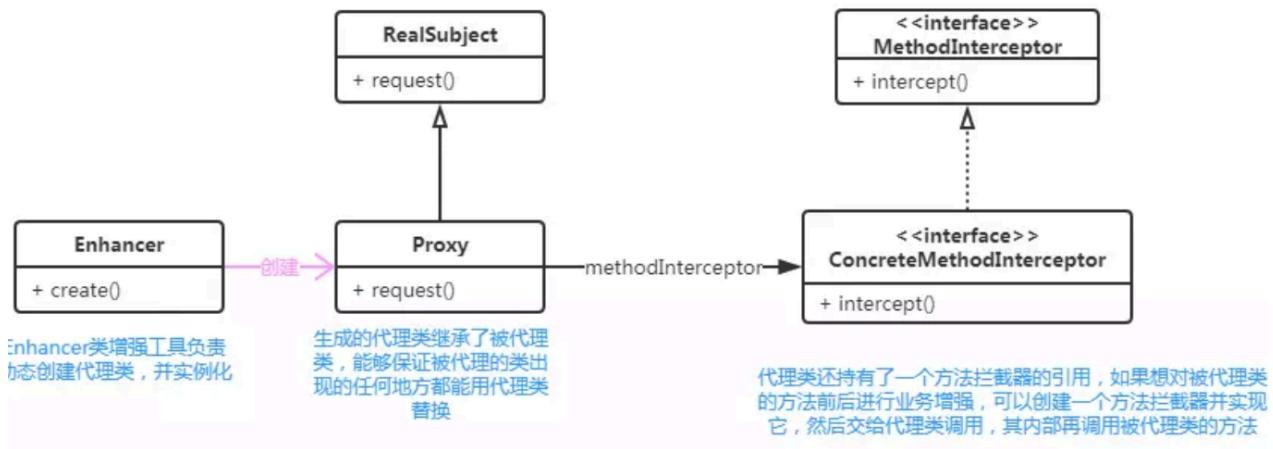
2. 实现InvocationHandler接口

```
public class ConcreteInvocationHandler implements InvocationHandler {  
  
    private Subject subject;  
  
    public ConcreteInvocationHandler(Subject subject) {  
        this.subject = subject;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {  
        return method.invoke(subject, args);  
    }  
}
```

3. 测试

```
public class JDKDynamicProxyTest {  
    public static void main(String[] args) {  
        Subject subject = new RealSubject();  
        InvocationHandler handler = new ConcreteInvocationHandler(subject);  
        Subject proxy =  
            (Subject)Proxy.newProxyInstance(RealSubject.class.getClassLoader(),  
                RealSubject.class.getInterfaces(), handler);  
        proxy.request();  
    }  
}
```

2. Cglib动态代理



1. 原有类

```

public class Target {
    public void request() {
        System.out.println("执行目标类的方法");
    }
}
  
```

2. 实现拦截器

```

public class TargetMethodInterceptor implements MethodInterceptor {
    @Override
    public Object intercept(Object obj, Method method, Object[] args,
                           MethodProxy proxy) throws Throwable {
        System.out.println("方法拦截增强逻辑-前置处理执行");
        Object result = proxy.invokeSuper(obj, args);
        System.out.println("方法拦截增强逻辑-后置处理执行");
        return result;
    }
}
  
```

3. 测试

```

public class CglibDynamicProxyTest {

    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();

        // 设置生成代理类的父类class对象
        enhancer.setSuperclass(Target.class);

        // 设置增强目标类的方法拦截器
        MethodInterceptor methodInterceptor = new TargetMethodInterceptor();
        enhancer.setMethodInterceptor(methodInterceptor);
        Target target = (Target) enhancer.create();
        target.request();
    }
}
  
```

```
enhancer.setCallback(methodInterceptor);

// 生成代理类并实例化
Target proxy = (Target) enhancer.create();

// 用代理类调用方法
proxy.request();
}

}
```

3. 静态代理和动态代理的区别

既然有了静态代理的方式，为什么还要使用动态代理的方式？

你会发现每个代理类只能为一个接口服务，这样程序开发中必然会产生许多的代理类所以我们就会想办法可以通过一个代理类完成全部的代理功能，那么我们就需要用动态代理。

静态代理和动态代理不同的特点？

普通代理模式，代理类Proxy的Java代码在JVM运行时就已经确定了，也就是在编码编译阶段就确定了Proxy类的代码。而动态代理是指在JVM运行过程中，动态的创建一个类的代理类，并实例化代理对象。因为实际的代理类是在运行时创建的。

4. JDK和CGLib实现方式对比

- 字节码创建方式：JDK动态代理通过JVM实现代理类字节码的创建，cglib通过ASM创建字节码。
- **JDK动态代理强制要求目标类必须实现了某一接口，否则无法进行代理。而CGLIB则要求目标类和目标方法不能是final的，因为CGLIB通过继承的方式实现代理。【InvocationHandler和MethodInterceptor】**
- CGLib不能对声明为final的方法进行代理，因为是通过继承父类的方式实现，如果父类是final的，那么无法继承父类。

5. JDK代理方式如何代理多个方法

另外，被代理类可以实现多个接口。从代理类代码中可以看到，代理类是通过InvocationHandler的invoke方法去实现被代理接口方法调用的。所以被代理对象实现了多个接口并且希望对不同接口实施不同的代理行为时，应该在ConcreteInvocationHandler类的invoke方法中，通过判断方法名来实现不同的接口的代理行为。

15. 多态定义&实现原理

```
package _00_Java_language.MutilStatus;

class Animal {
    int num = 10;
    static int age = 20;

    public void eat(){
        System.out.println("Animal eat");
    }
}

interface Eat {
    void eat();
}
```

```
        System.out.println("动物吃饭");
    }

    public static void sleep(){
        System.out.println("动物睡觉");
    }

    public void run(){
        System.out.println("动物奔跑");
    }
}

class Cat extends Animal{
    int num = 80;
    static int age = 90;
    String name = "tomCat";

    public void eat(){
        System.out.println("猫在吃饭");
    }

    public static void sleep(){
        System.out.println("猫在睡觉");
    }

    public void catchMouse(){
        System.out.println("猫在抓老鼠");
    }
}

public class Test{
    public static void main(String[] args) {
        Animal animal = new Cat(); //父类引用指向子类对象

        animal.eat(); //子类 猫在吃饭
        animal.sleep(); //父类 动物在睡觉
        animal.run(); //父类 动物在奔跑

        System.out.println(animal.num); //父类 10
        System.out.println(animal.age); //父类 20
    }
}
```

成员变量

编译看左边(父类),运行看左边(父类)

成员方法

编译看左边(父类), 运行看右边(子类)。动态绑定

静态方法

编译看左边(父类), 运行看左边(父类)。

(静态和类相关, 算不上重写, 所以, 访问还是左边的)

只有非静态的成员方法,编译看左边,运行看右边

- 多态的条件
 - 父类引用指向子类实例化对象
- 多态的限制
 - 不能使用子类中特有的方法和成员变量

16. 四种引用

几种引用需要进行实践, 才能得出更加深入的结论

1. 强引用(StrongReference)

如果一个对象具有强引用, 那垃圾回收器绝不会回收它。当内存空间不足, Java虚拟机宁愿抛出 OutOfMemoryError 错误, 使程序异常终止, 也不会靠随意回收具有强引用的对象来解决内存不足的问题。

值得注意的是: 如果想中断或者回收强引用对象, 可以显式地将引用赋值为 null, 这样的话JVM就会在合适的时间, 进行垃圾回收。

```
String[] arr = new String[]{"a", "b", "c"};
```

2. 软引用(SoftReference)

如果一个对象只具有软引用, 则内存空间足够, 垃圾回收器就不会回收它; 如果内存空间不足了, 就会回收这些对象的内存。只要垃圾回收器没有回收它, 该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。 软引用可以和一个引用队列 (ReferenceQueue) 联合使用, 如果软引用所引用的对象被垃圾回收器回收, Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

```

//示例1
SoftReference<String[]> softBean = new SoftReference<String[]>(new String[]
{"a", "b", "c"});

//示例2
ReferenceQueue<String[]> referenceQueue = new ReferenceQueue<String[]>();
SoftReference<String[]> softBean = new SoftReference<String[]>(new String[]
{"a", "b", "c"}, referenceQueue);

```

3. 弱引用(WeakReference)

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用(Phantom Reference)

虚引用是最弱的一种引用关系，如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收，在JDK1.2之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个 null，也就是说将永远无法通过虚引用获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

参考：<https://www.cnblogs.com/liyutian/p/9690974.html>

17. 动态分派&静态分派

1. 静态分派

```

public class StaticDispatch {
    static abstract class Human{
    }
    static class Man extends Human{
    }
    static class Woman extends Human{
    }
    public static void sayHello(Human guy){
        System.out.println("hello,guy!");
    }
    public static void sayHello(Man guy){
        System.out.println("hello,gentlemen!");
    }
    public static void sayHello(Woman guy){
        System.out.println("hello,lady!");
    }

    public static void main(String[] args) {
        Human man=new Man();
    }
}

```

```

Human woman=new Woman();
sayHello(man); //输出hello,guy
sayHello(woman); //输出hello,guy
}
}

```

- 重载：方法名相同，参数列表不同
- 编译器在**重载**时是通过**参数的静态类型**而不是实际类型作为判定的依据。并且静态类型在编译期可知，因此，编译阶段，Javac编译器会根据**参数的静态类型**决定使用哪个重载版本。

2. 动态分派

1. Override实现多态

```

public class DynamicDispatch {
    static abstract class Human{
        protected abstract void sayHello();
    }
    static class Man extends Human{
        @Override
        protected void sayHello() {
            System.out.println("man say hello!");
        }
    }
    static class Woman extends Human{
        @Override
        protected void sayHello() {
            System.out.println("woman say hello!");
        }
    }
    public static void main(String[] args) {

        Human man=new Man();
        Human woman=new Woman();
        man.sayHello(); //输出 man say hello
        woman.sayHello(); //输出 woman say hello

        man=new Woman();
        man.sayHello();
    }
}

```

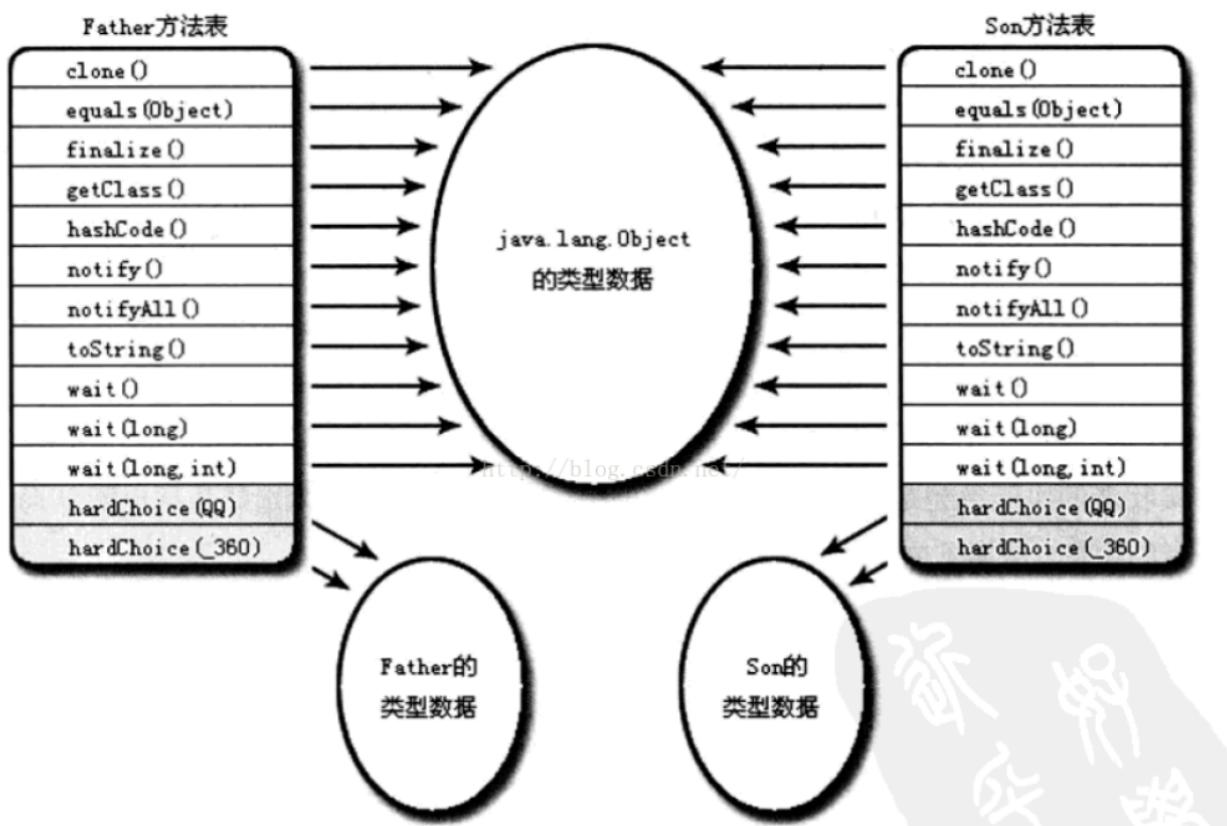
显然，这里不可能再根据静态类型来决定，因为静态类型同样是Human的两个变量man和woman在调用sayHello()方法时执行了不同的行为，并且变量man在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同，Java虚拟机是如何根据实际类型来分派方法执行版本的呢？我们从invokevirtual指令的多态查找过程开始说起，invokevirtual指令的运行时解析过程大致分为以下几个步骤：

1、找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C。2、如果在类型C中找到与常量中的描述符和简单名称相符合的方法，然后进行访问权限验证，如果验证通过则返回这个方法的直接引用，查找过程结束；如果验证不通过，则抛出java.lang.IllegalAccessError异常。3、否则未找到，就按照继承关系从下往上依次对类型C的各个父类进行第2步的搜索和验证过程。4、如果始终没有找到合适的方法，则跑出java.lang.AbstractMethodError异常。

由于`invokevirtual`指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的`invokevirtual`指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是Java语言方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

2. JVM实现原理

Java虚拟机是如何实现动态分派功能的？



虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都是指向父类的实际入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实际版本的入口地址。

为了程序实现上的方便，具有相同签名的方法，在父类、子类的虚方法表中具有一样的索引序号，这样当类型变换时，仅仅需要变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需要的入口地址。

方法表一般在类加载阶段的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。

18. Java是传值还是传引用

1. 基本数据类型直接拷贝，对原有变量不会有影响
2. 对于对象，需要判断是否是同一个引用，以及对堆内存是否发生影响

```
package _00_Java_language.passByValue;

public class Test3 {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello ");
        System.out.println("Before change, sb = " + sb); //Hello
        changeData(sb);
        System.out.println("After changeData(n), sb = " + sb); //Hello
    }

    public static void changeData(StringBuffer strBuf) { //这里的strBuf是原有sb的
        strBuf = new StringBuffer("Hi "); //strBuf指向了其他的位置
        strBuf.append("World!"); //strBuf指向位置进行了修改
    }
}
```

002 I/O

首先，传统的 `java.io` 包，它基于流模型实现，提供了我们最熟知的一些 IO 功能，比如 `File` 抽象、输入输出流等。交互方式是同步、阻塞的方式，也就是说，在读取输入流或者写入输出流时，在读、写动作完成之前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。

`java.io` 包的好处是代码比较简单、直观，缺点则是 IO 效率和扩展性存在局限性，容易成为应用性能的瓶颈。

很多时候，人们也把 `java.net` 下面提供的部分网络 API，比如 `Socket`、`ServerSocket`、`HttpURLConnection` 也归类到同步阻塞 IO 类库，因为网络通信同样是 IO 行为。

第二，在 Java 1.4 中引入了 NIO 框架 (`java.nio` 包)，提供了 `Channel`、`Selector`、`Buffer` 等新的抽象，可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层的高性能数据操作方式。

第三，在 Java 7 中，NIO 有了进一步的改进，也就是 NIO 2，引入了异步非阻塞 IO 方式，也有很多人叫它 AIO (Asynchronous IO)。异步 IO 操作基于事件和回调机制，可以简单理解为，应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

1. BIO 【Blocking IO】

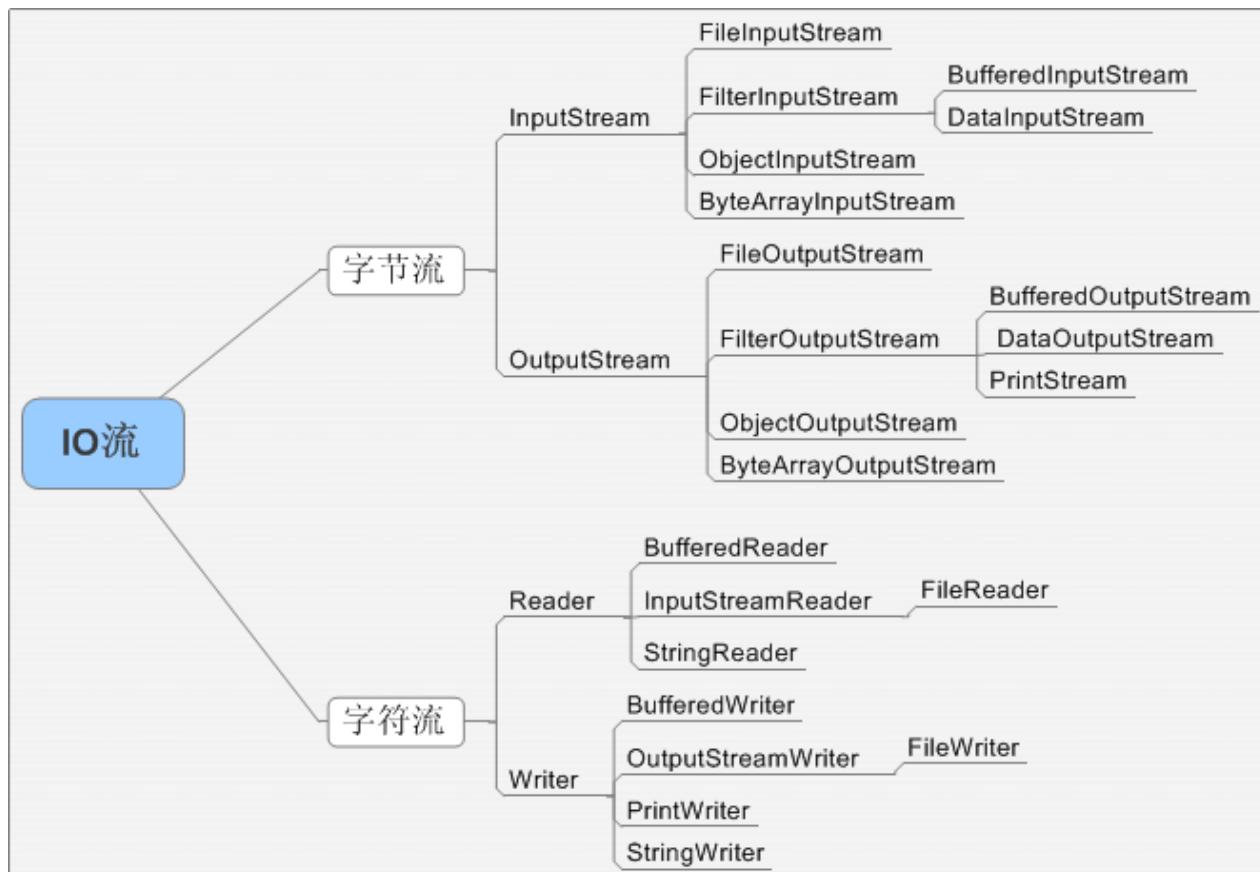
1. 字节流和字符流有什么区别

在应用中，经常要完全是字符的一段文本输出去或读进来，用字节流可以吗？计算机中的一切最终都是二进制的字节形式存在。对于“中国”这些字符，首先要得到其对应的字节，然后将字节写入到输出流。读取时，首先读到的是字节，可是我们要把它显示为字符，我们需要将字节转换成字符。

由于这样的需求很广泛，人家专门提供了字符流的包装类。底层设备永远只接受字节数据，有时候要写字符串到底层设备，需要将字符串转成字节再进行写入。字符流是字节流的包装，字符流则是直接接受字符串，它内部将串转成字节，再写入底层设备，这为我们向 IO 设别写入或读取字符串提供了一点点方便。

字符向字节转换时，要注意编码的问题，因为字符串转成字节数组，其实是转成该字符的某种编码的字节形式，读取也是反之的道理。

- 字节流 【byte = 8bit】
- 字符流 【char = 2 * byte = 16bit】



2. AIO 【Asynchronous IO】

3. NIO 【New IO/Non-Blocking IO】

总结：http://www.jasongj.com/java/nio_reactor/

美团技术文章：<https://tech.meituan.com/2016/11/04/nio.html>

2. Unix IO模型

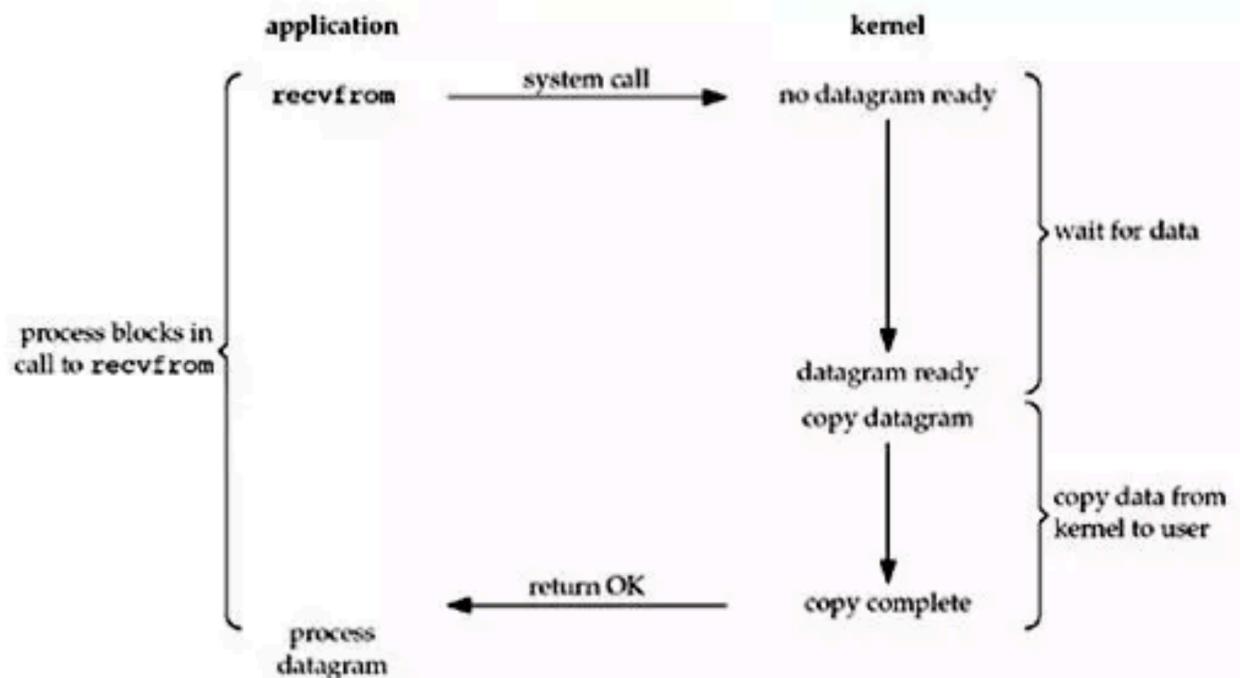
Unix 下共有五种 I/O 模型：

- 阻塞 I/O

- 非阻塞 I/O
- I/O 多路复用 (select和poll)
- 信号驱动 I/O (SIGIO)
- 异步 I/O (Posix.1的aio_系列函数)

1. 阻塞IO

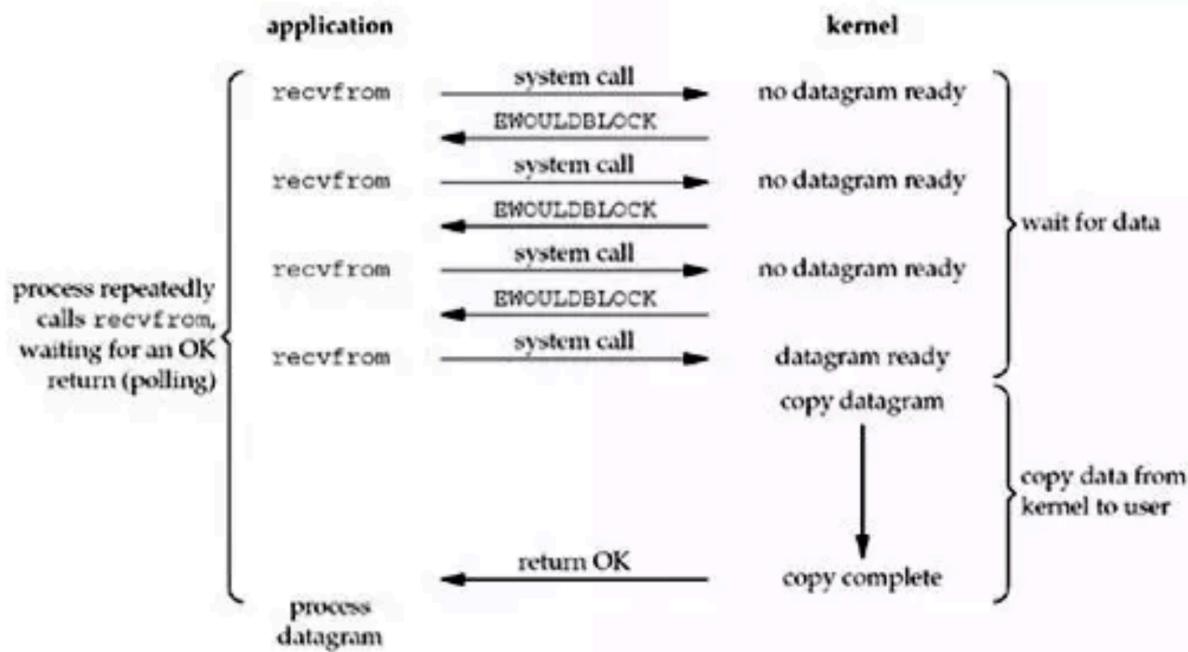
Figure 6.1. Blocking I/O model.



当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候kernel就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。所以，blocking IO的特点就是在IO执行的两个阶段都被block了。

2. 非阻塞IO

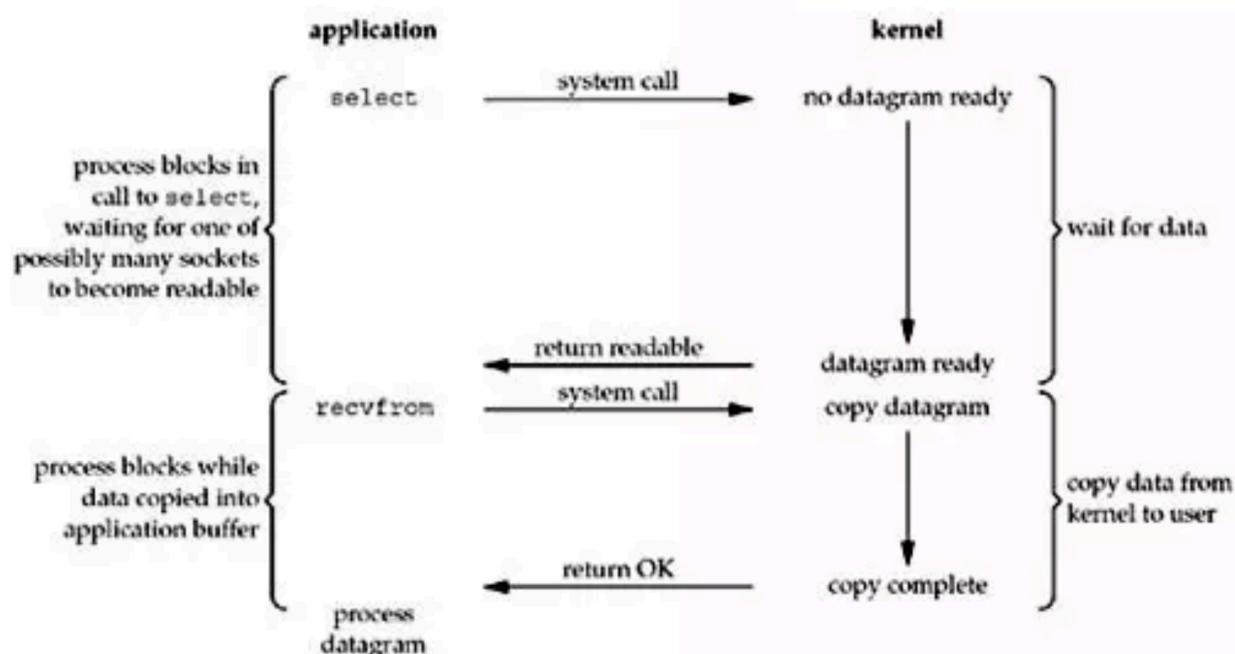
Figure 6.2. Nonblocking I/O model.



3. IO多路复用

从图中可以看出，当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。所以，用户进程其实是需要不断的主动询问kernel数据好了没有。

Figure 6.3. I/O multiplexing model.



当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。（多说一句。所以，如果处理的连接数不是很高的话，使用select epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

- IO多路复用出现是为了解决什么问题？
 - 由于进程的执行过程是线性的(也就是顺序执行),当我们调用低速系统I/O(read,write,accept等等),进程可能阻塞,此时进程就阻塞在这个调用上,不能执行其他操作.阻塞很正常.
 - 接下来考虑这么一个问题:一个服务器进程和一个客户端进程通信,服务器端read(sockfd1,bud,bufsize),此时客户端进程没有发送数据,那么read(阻塞调用)将阻塞,直到客户端调用write(sockfd,bud,size)发来数据.在一个客户和服务器通信时这没什么问题.
 - 当多个客户与服务器通信时当多个客户与服务器通信时,若服务器阻塞于其中一个客户sockfd1,当另一个客户的数据到达套接字sockfd2时,服务器不能处理,仍然阻塞在read(sockfd1,...)上;此时问题就出现了,不能及时处理另一个客户的服务,咋么办?

3. NIO

1. NIO作用

NIO【Non-Blocking IO】是面向缓冲区的。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。

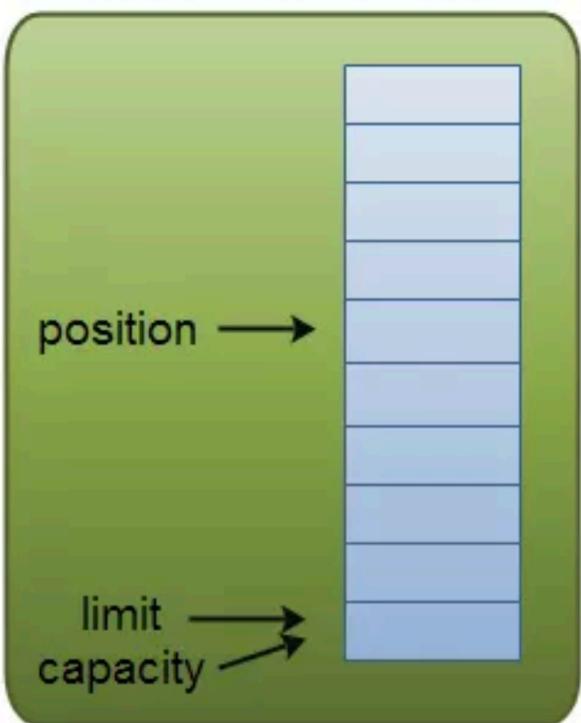
Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此，一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。

NIO是可以做到用一个线程来处理多个操作的。假设有10000个请求过来,根据实际情况，可以分配50或者100个线程来处理。不像之前的阻塞IO那样，非得分配10000个。

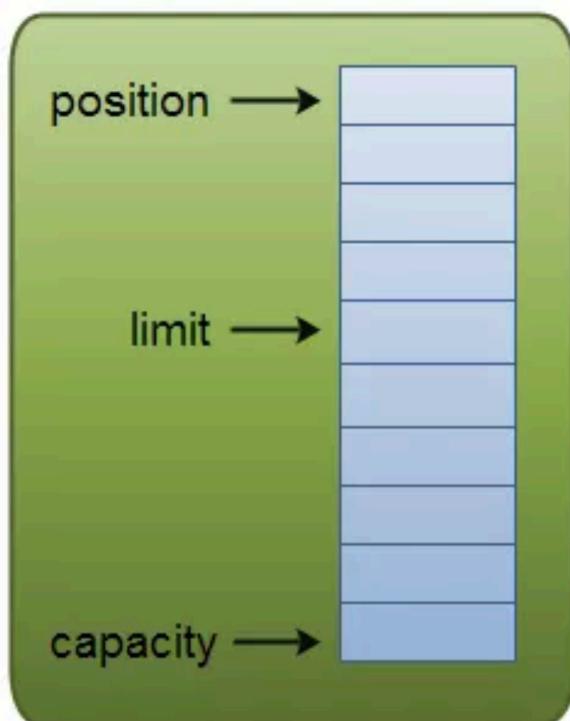
2. NIO组成

1. Buffer

Buffer - Write Mode



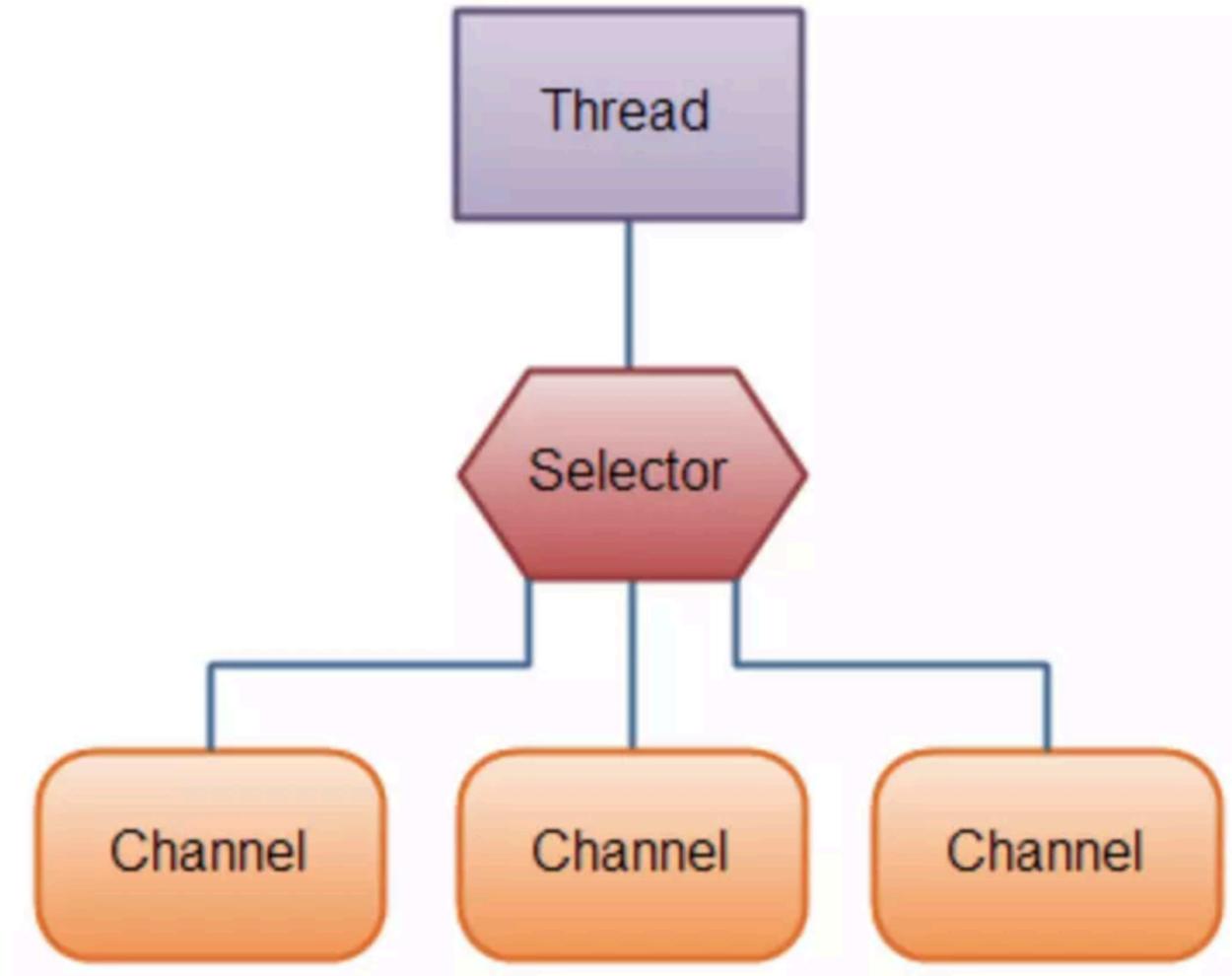
Buffer - Read Mode



在对Buffer进行读/写的过程中，position会往后移动，而 limit 就是 position 移动的边界。由此不难想象，在对**Buffer**进行写入操作时，**limit**应当设置为**capacity**的大小，而对**Buffer**进行读取操作时，**limit**应当设置为数据的实际结束位置。（注意：将Buffer数据写入通道是Buffer 读取操作，从通道读取数据到Buffer是Buffer 写入操作）

2. Channel

3. Selector



- 为什么要使用Selector

- 前文说了，如果用阻塞I/O，需要多线程（浪费内存），如果用非阻塞I/O，需要不断重试（耗费CPU）。Selector的出现解决了这尴尬的问题，非阻塞模式下，通过Selector，我们的线程只为已就绪的通道工作，不用盲目的重试了。比如，当所有通道都没有数据到达时，也就没有Read事件发生，我们的线程会在select()方法处被挂起，从而让出了CPU资源。

3. NIO和IO有什么区别

1. 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。

2. 阻塞IO和非阻塞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，所以直至数据变得可以读取之前，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）。

4. NIO案例-拷贝文件

Buffer是一个对象，它包含一些要写入或者读到Stream对象的。应用程序不能直接对 Channel 进行读写操作，而必须通过 Buffer 来进行，即 Channel 是通过 Buffer 来读写数据的。

在NIO中，所有的数据都是用Buffer处理的，它是NIO读写数据的中转池。Buffer实质上是一个数组，通常是一个字节数据，但也可以是其他类型的数组。但一个缓冲区不仅仅是一个数组，重要的是它提供了对数据的结构化访问，而且还可以跟踪系统的读写进程。

```
public static void copyFileUseNIO(String src, String dst) throws IOException{
    //声明源文件和目标文件
    FileInputStream fi=new FileInputStream(new File(src));
    FileOutputStream fo=new FileOutputStream(new File(dst));
    //获得传输通道channel
    FileChannel inChannel=fi.getChannel();
    FileChannel outChannel=fo.getChannel();
    //获得容器buffer
    ByteBuffer buffer=ByteBuffer.allocate(1024);
    while(true){
        //判断是否读完文件
        int eof =inChannel.read(buffer);
        if(eof==-1){
            break;
        }
        //重设一下buffer的position=0, limit=position
        buffer.flip();
        //开始写
        outChannel.write(buffer);
        //写完要重置buffer, 重设position=0,limit=capacity
        buffer.clear();
    }
    inChannel.close();
    outChannel.close();
    fi.close();
    fo.close();
}
```

4. 相关问题

1. 阻塞/非阻塞和异步/同步的区别

1. 同步与异步

【同步和异步：线程的调用是1.调用就得到结果还是 2.立即返回，有结果的时候再通知】

同步和异步关注的是消息通信机制(synchronous communication/ asynchronous communication) 所谓同步，就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由调用者主动等待这个调用的结果。

而异步则是相反，【调用】在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调用。

典型的异步编程模型比如Node.js

举个通俗的例子：你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，“我查一下”，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

2. 阻塞与非阻塞

【阻塞和非阻塞：线程在得到调用结果之前能不能干别的事情】

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

还是上面的例子，你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了，当然你也要偶尔过几分钟check一下老板有没有返回结果。在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

2. select/poll/epoll

- 使用select以后最大的优势是用户可以在一个线程内同时处理多个socket的IO请求。
- poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。
- epoll是Linux内核为处理大批量文件描述符而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下系统的CPU利用率。原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。

参考：<https://www.jianshu.com/p/397449cadc9a>

参考：<https://baijiahao.baidu.com/s?id=1611547498841608701&wfr=spider&for=pc>

3. 解释一下什么是I/O多路复用

- 发明它的原因，是尽量多的提高服务器的吞吐能力。
- 重要的事情再说一遍：I/O multiplexing 这里面的 multiplexing 指的其实是在单个线程通过记录跟踪每一个Sock(I/O流)的状态(对应空管塔里面的Fight progress strip槽)来同时管理多个I/O流。



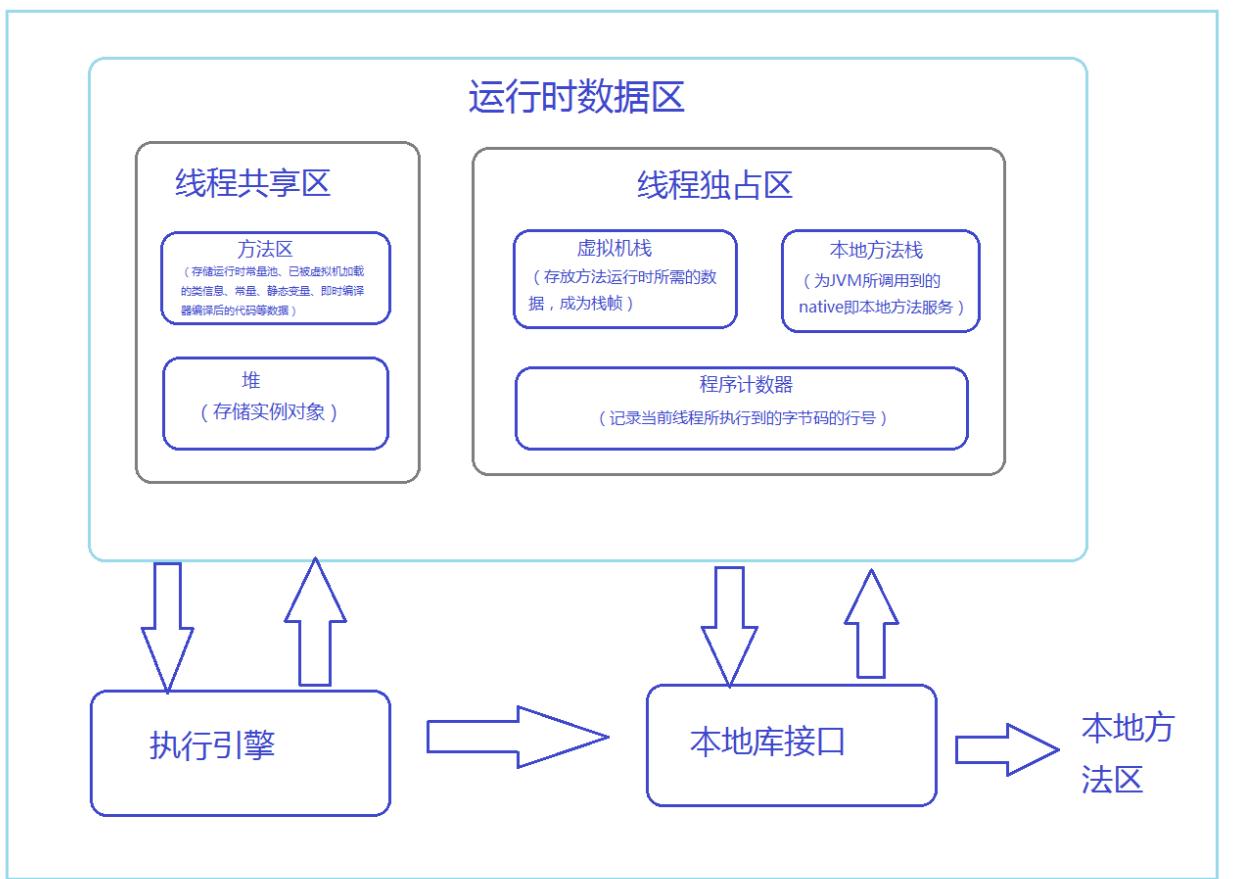
003 JVM

1. JVM内存的构成。★★★★★

关于JVM进阶知识：<https://blog.csdn.net/sunhuaiqiang1/column/info/15328>

内存分区是一个特别基础的概念，不过要结合Java实际编程来分析，理解更加深刻。

运行时数据区（Runtime Data）包含哪些部分？



<https://blog.csdn.net/qq906627950>

1. 栈存放的是什么

```
int i = 2; //基本数据类型  
String s = new String("abc"); //s引用
```

2. 堆存放的是什么

new出来的对象本身。

3. 调用方法的结果【传值还是传参数的问题】

2. 谈一谈GC回收算法。★★★★★

在学习GC的时候保持一条主线：

- 要回收什么
- 如何判断对象是否需要回收
- 如何回收对象[占用的内存区域]

一、要回收哪些区域

在JVM内存模型中，有三个是不需要进行垃圾回收的：程序计数器、JVM栈、本地方法栈。因为它们的生命周期是和线程同步的，随着线程的销毁，它们占用的内存会自动释放，所以只有方法区和堆需要进行GC

引用计数法：(无法解决循环引用的问题)

在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（ $a = b$, 则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

二、如何判断对象是否需要回收

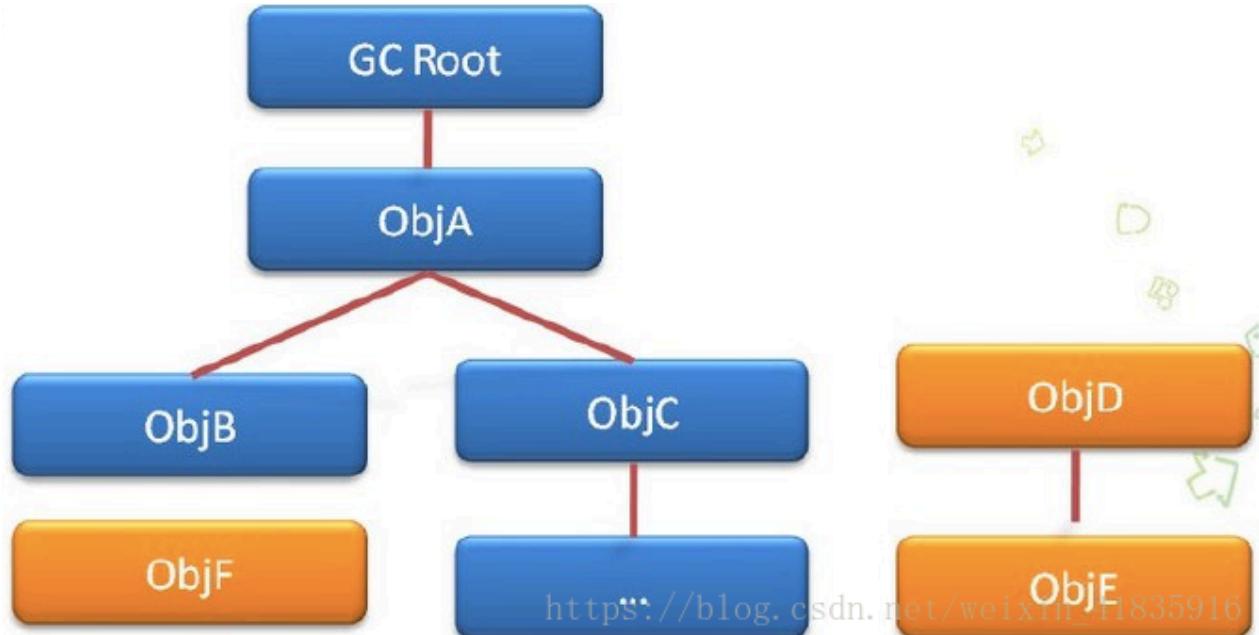
可达性分析算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点**GC ROOT**开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没被引用到的节点，即无用的节点，无用的节点将会被判定为是可回收的对象。

在Java语言中，可作为GC Roots的对象包括下面几种： a) 虚拟机栈中引用的对象（栈帧中的本地变量表）；

b) 方法区中类静态属性引用的对象；

c) 方法区中常量引用的对象；

d) 本地方法栈中JNI (Native方法) 引用的对象。



备注：

方法区如何判断是否需要回收 方法区主要回收的内容有：废弃常量和无用的类。对于废弃常量也可通过引用的可达性来判断，但是对于无用的类则需要同时满足下面3个条件：①该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例；②加载该类的ClassLoader已经被回收；

③该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

三、回收算法

(1)标记-清除算法（Mark-Sweep）

标记-清除算法采用从根集合（GC Roots）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，此算法一般没有虚拟机采用。

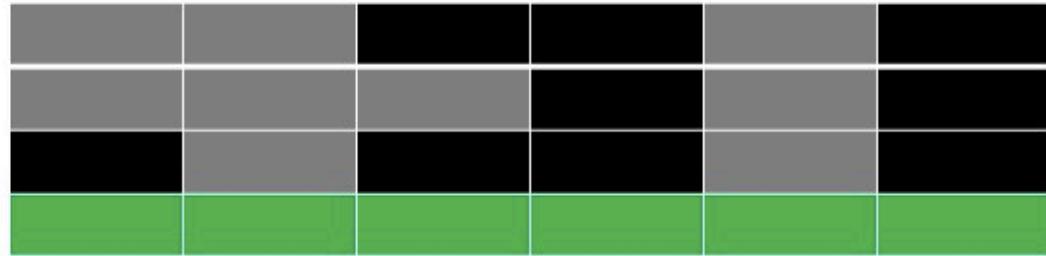
标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。

优点1：解决了循环引用的问题 优点2：与复制算法相比，不需要对象移动，效率较高，而且还不需要额外的空间 不足1：每个活跃的对象都要进行扫描，而且要扫描两次，效率较低，收集暂停的时间比较长。

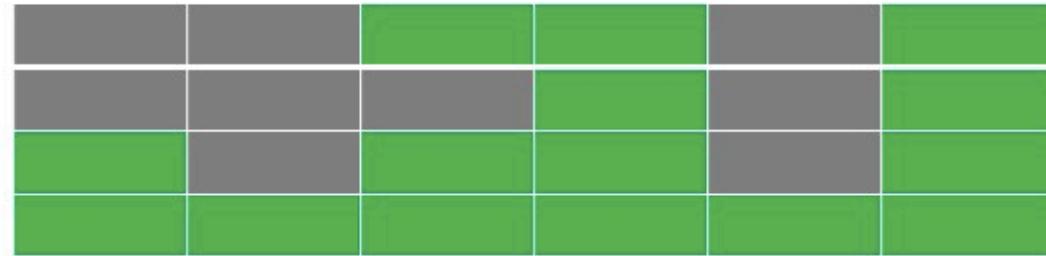
不足2：产生不连续的内存碎片

备注：CMS垃圾回收器

标记后



清除后



存活对象

未使用 https://blog.csdn.net/weixin_41835916 可回收

(2) 复制算法 (Copying)

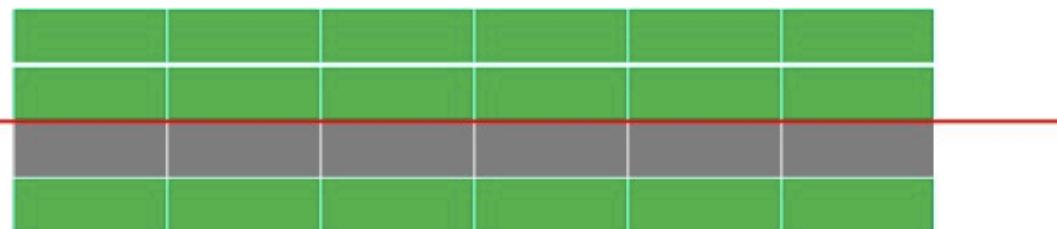
将内存分成两块容量大小相等的区域，每次只使用其中一块，当这一块内存用完了，就将所有存活对象复制到另一块内存空间，然后清除前一块内存空间。这样一来就不容易出现内存碎片的问题。
1、复制的代价较高，所以适合新生代，因为新生代的对象存活率较低，需要复制的对象较少；

2、需要双倍的内存空间，而且总是有一块内存空闲，浪费空间

备注：Minor GC



回收后



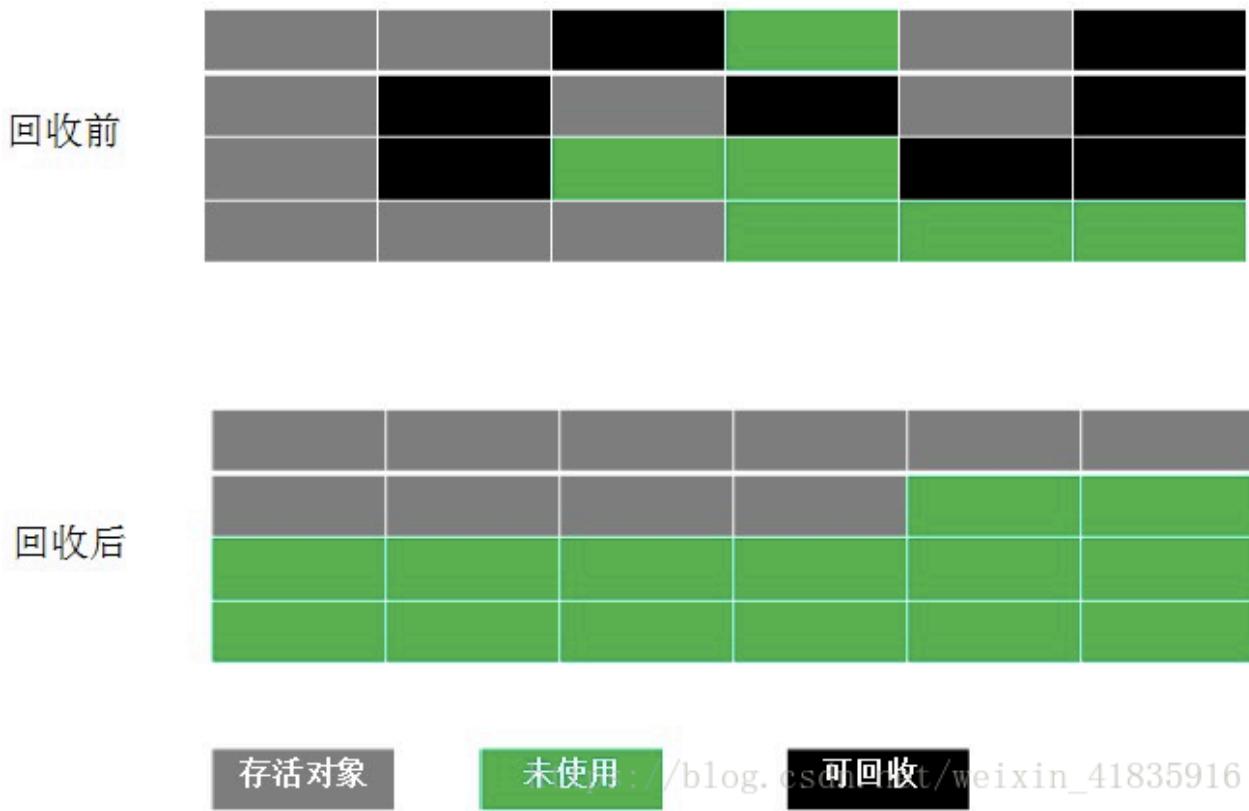
存活对象

未使用

https://blog.csdn.net/weixin_41835916 可回收

(3) 标记-整理算法 (Mark-Compact)

在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。特点：不会产生内存碎片，但是依旧移动对象的成本。

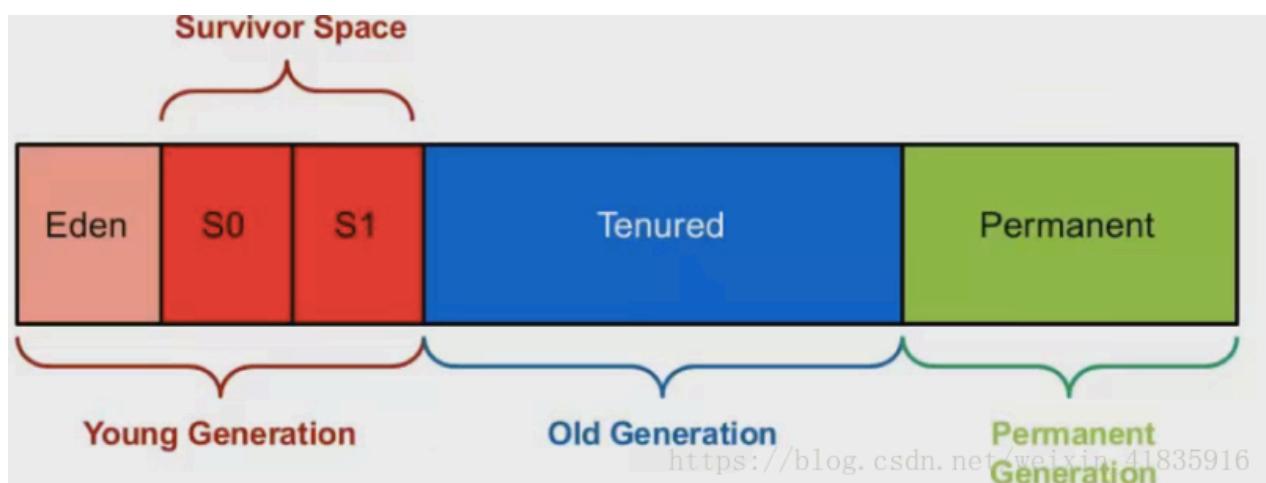


备注：FullGC

(4)分代搜集算法 (Generational Collection)

分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanent Generation）。老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

新生代：Eden、from Survivor Space and to Survivor Space



[1]新生代回收算法

包含有Eden、form survivor space、to survivor space三个区，绝大多数最新被创建的对象会被分配到这里，大部分对象在创建之后会变得很快不可达。

① 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

② 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空，如此往复。

③ 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收。

④ 新生代发生的GC也叫做Minor GC，Minor GC发生频率比较高(不一定等Eden区满了才触发)。

[2]老年代回收算法

① 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

② 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

一般来说，老年代使用**Mark-Sweep**算法或者**Mark-Compact**算法，因为对象存活率高，不能使用额外空间担保。

Java 中的堆也是 GC 收集垃圾的主要区域。GC 分为两种：Minor GC、Full GC (或称为 Major GC)。

Minor GC 是发生在新生代中的垃圾收集动作，所采用的是复制算法。

新生代几乎是所有 Java 对象出生的地方，即 Java 对象申请的内存以及存放都是在这个地方。Java 中的大部分对象通常不需长久存活，具有朝生夕灭的性质。

当一个对象被判定为 "死亡" 的时候，GC 就有责任来回收掉这部分对象的内存空间。新生代是 GC 收集垃圾的频繁区域。

当对象在 Eden (包括一个 Survivor 区域，这里假设是 from 区域) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即 to 区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即 from 区域)，并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 (默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定)，这些对象就会成为老年代。

但这也不是一定的，对于一些较大的对象 (即需要分配一块较大的连续内存空间) 则是直接进入到老年代。

Full GC 是发生在老年代的垃圾收集动作，所采用的是标记-清除算法。

现实的生活中，老年代的人通常会比新生代的人 "早死"。堆内存中的老年代(old)不同于这个，老年代里面的对象几乎个个都是在 Survivor 区域中熬过来的，它们是不会那么容易就 "死掉" 了的。因此，Full GC 发生的次数不会有 Minor GC 那么频繁，并且做一次 Full GC 要比进行一次 Minor GC 的时间更长。

另外，标记-清除算法收集垃圾的时候会产生许多的内存碎片 (即不连续的内存空间)，此后需要为较大的对象分配内存空间时，若无法找到足够的连续的内存空间，就会提前触发一次 GC 的收集动作。

备注：永久代从JDK8开始移出，新的叫做元数据。

备注2：关于JVM新生代、老年代：<http://ju.outofmemory.cn/entry/346964>

3. 谈一谈MinorGC和FullGC ★★★★

MinorGC：同2—新生代回收算法

FullGC：同2—老年代回收算法

附加问题：MinorGC和FullGC的触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

- (1) System.gc()方法的调用 【此方法的调用是建议JVM进行Full GC,虽然只是建议而非一定】
- (2) 老年代空间不足
- (3) 方法区空间不足
- (4) 通过Minor GC后进入老年代的平均大小大于老年代的可用内存
- (5) 由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

5. 为什么要划分成年轻代和老年代★★★★

分代的唯一理由就是优化GC性能

- 如果没有分代，所有的对象都在一块，GC的时要找到哪些对象是没用的，这样就会对堆的所有区域进行扫描。而我们的很多对象都是朝生夕死的。
- 如果分代的话，把新创建的对象放到某一地方，当GC的时先把这块存“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。

6. 年轻代—>年轻代为什么被划分成 eden、survivor 区域&年轻代为什么采用的是复制算法 ★★

这是一个非常有意思的问题，分析的过程也比较好玩

- 如果没有survivor区，那么很容易触发Major GC，影响程序效率
- 如果只有一个survivor区，那么容易在eden区域和survivor区域中产生内存碎片，影响内存分配

如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Major GC（因为Major GC一般伴随着Minor GC，也可以看做触发了Full GC）。老年代的内存空间远大于新生代，进行一次Full GC消耗的时间比Minor GC长得多。你也许会问，执行时间长有什么坏处？频发的Full GC消耗的时间是非常可观的，这一点会影响大型程序的执行和响应速度，更不要说某些连接会因为超时发生连接错误了。

好，那我们来想想在没有Survivor的情况下，有没有什么解决办法，可以避免上述情况：

方案	优点	缺点
增加老年代空间	更多存活对象才能填满老年代。 降低Full GC频率	随着老年代空间加大，一旦发生Full GC，执行所需要的时间更长
减少老年代空间	Full GC所需时间减少	老年代很快被存活对象填满，Full GC频率增加

显而易见，没有Survivor的话，上述两种解决方案都不能从根本上解决问题。

我们可以得到第一条结论：**Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。**

设置两个Survivor区最大的好处就是解决了碎片化，下面我们来分析一下。

为什么一个Survivor区不行？第一部分中，我们知道必须设置Survivor区。假设现在只有一个survivor区，我们来模拟一下流程：刚刚新建的对象在Eden中，一旦Eden满了，触发一次Minor GC，Eden中的存活对象就会被移动到Survivor区。这样继续循环下去，下一次Eden满了的时候，问题来了，此时进行Minor GC，Eden和Survivor各有一些存活对象，如果此时把Eden区的存活对象硬放到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化。

那么，顺理成章的，应该建立两块Survivor区，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）。S0和Eden被清空，然后下一轮S0与S1交换角色，如此循环往复。如果对象的复制次数达到16次，该对象就会被送到老年代中。

7. 老年代—>老年代为什么采用的是标记清除、标记整理算法 ★★

老年代不使用复制算法的原因就是复制算法的缺点：

- 空间成本高，需要双倍空间
- 复制的时间成本高

8. 堆外内存—>什么是堆外内存，如何被回收 ★★★★

10. 静态变量和实例变量的区别 ★★★★

需要结合Java类加载过程进行说明

一、实例变量

也叫对象变量、类成员变量；从属于类由类生成对象时，才分配存储空间，各对象间的实例变量互不干扰，能通过对象的引用来访问实例变量。但在Java多线程中，实例变量是多个线程共享资源，要注意同步访问时可能出现的问题。

二、类变量

也叫静态变量，是一种比较特殊的实例变量，用static关键字修饰；一个类的静态变量，所有由这类生成的对象都共用这个类变量，类装载时就分配存储空间。一个对象修改了变量，则所以对象中这个变量的值都会发生改变。

11. 类加载机制



当我们的Java代码编译完成后，会生成对应的 class 文件。接着我们运行 `java Demo` 命令的时候，我们其实是启动了JVM 虚拟机执行 class 字节码文件的内容。而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

- 加载：将Class文件读入内存
- 链接
 - 验证：验证字节码是否合法
 - 准备：static修饰的变量分配内存，赋值为零[final static除外]
 - 解析：替换常亮池的符号引用为直接引用
- 初始化：类变量和静态变量赋值



1. 加载-class文件到java.lang.Class对象

加载指的是将类的class文件读入到内存，并为之创建一个java.lang.Class对象，也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

类的加载由类加载器完成，类加载器通常由JVM提供，这些类加载器也是前面所有程序运行的基础，JVM提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承ClassLoader基类来创建自己的类加载器。

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源。

- 从本地文件系统加载class文件，这是前面绝大部分示例程序的类加载方式。
- 从JAR包加载class文件，这种方式也是很常见的，前面介绍JDBC编程时用到的数据库驱动类就放在JAR文件中，JVM可以从JAR文件中直接加载该class文件。

- 通过网络加载class文件。
- 把一个Java源文件动态编译，并执行加载。

类加载器通常无须等到“首次使用”该类时才加载该类，Java虚拟机规范允许系统预先加载某些类。

2.验证-[e.g 方法缺少返回值]

当JVM加载完Class字节码文件并在方法区创建对应的Class对象之后，JVM便会启动对该字节码流的校验，只有符合JVM字节码规范的文件才能被JVM正确执行。这个校验过程大致可以分为下面几个类型：

- **JVM规范校验。** JVM会对字节流进行文件格式校验，判断其是否符合JVM规范，是否能被当前版本的虚拟机处理。例如：文件是否是以`0x cafe bene`开头，主次版本号是否在当前虚拟机处理范围之内等。
- **代码逻辑校验。** JVM会对代码组成的数据流和控制流进行校验，确保JVM运行该字节码文件后不会出现致命错误。例如一个方法要求传入int类型的参数，但是使用它的时候却传入了一个String类型的参数。一个方法要求返回String类型的结果，但是最后却没有返回结果。代码中引用了一个名为Apple的类，但是你实际上却没有定义Apple类。

3.准备-static变量分配内存

当完成字节码文件的校验之后，JVM便会开始为类变量分配内存并初始化。这里需要注意两个关键点，即内存分配的对象以及初始化的类型。

- **内存分配的对象。** Java中的变量有「类变量」和「类成员变量」两种类型，「类变量」指的是被`static`修饰的变量，而其他所有类型的变量都属于「类成员变量」。在准备阶段，JVM只会为「类变量」分配内存，而不会为「类成员变量」分配内存。「类成员变量」的内存分配需要等到初始化阶段才开始。

例如下面的代码在准备阶段，只会为`factor`属性分配内存，而不会为`website`属性分配内存。

```
public static int factor = 3;
public String website = "www.cnblogs.com/chanshuyi";
```

- **初始化的类型。** 在准备阶段，JVM会为类变量分配内存，并为其初始化。但是这里的初始化指的是为变量赋予Java语言中该数据类型的零值，而不是用户代码里初始化的值。

例如下面的代码在准备阶段之后，`sector`的值将是0，而不是3。

```
public static int sector = 3;
```

但如果一个变量是常量（被`static final`修饰）的话，那么在准备阶段，属性便会被赋予用户希望的值。例如下面的代码在准备阶段之后，`number`的值将是3，而不是0。

```
public static final int number = 3;
```

4.解析-解析引用

当通过准备阶段之后，JVM 针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类引用进行解析。这个阶段的主要任务是将其在常量池中的符号引用替换成直接其在内存中的直接引用。

其实这个阶段对于我们来说也是几乎透明的，了解一下就好。

5. 初始化-static变量分配指定值

到了初始化阶段，用户定义的 Java 程序代码才真正开始执行。

①声明类变量是指定初始值 ②使用静态代码块为类变量指定初始值

在这个阶段，JVM 会根据语句执行顺序对类对象进行初始化，一般来说当 JVM 遇到下面 5 种情况的时候会触发初始化：

- 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
- 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
- 当使用 JDK1.7 动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getstatic,REF_putstatic,REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

5. 使用

当 JVM 完成初始化阶段之后，JVM 便开始从入口方法开始执行用户的程序代码。这个阶段也只是了解一下就可以。

6. 卸载

当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存。这个阶段也只是了解一下就可以。

其他：

实际上Java代码编译成字节码之后，是没有构造方法的概念的，只有类初始化方法 和 对象初始化方法。那么这两个方法是怎么来的呢？

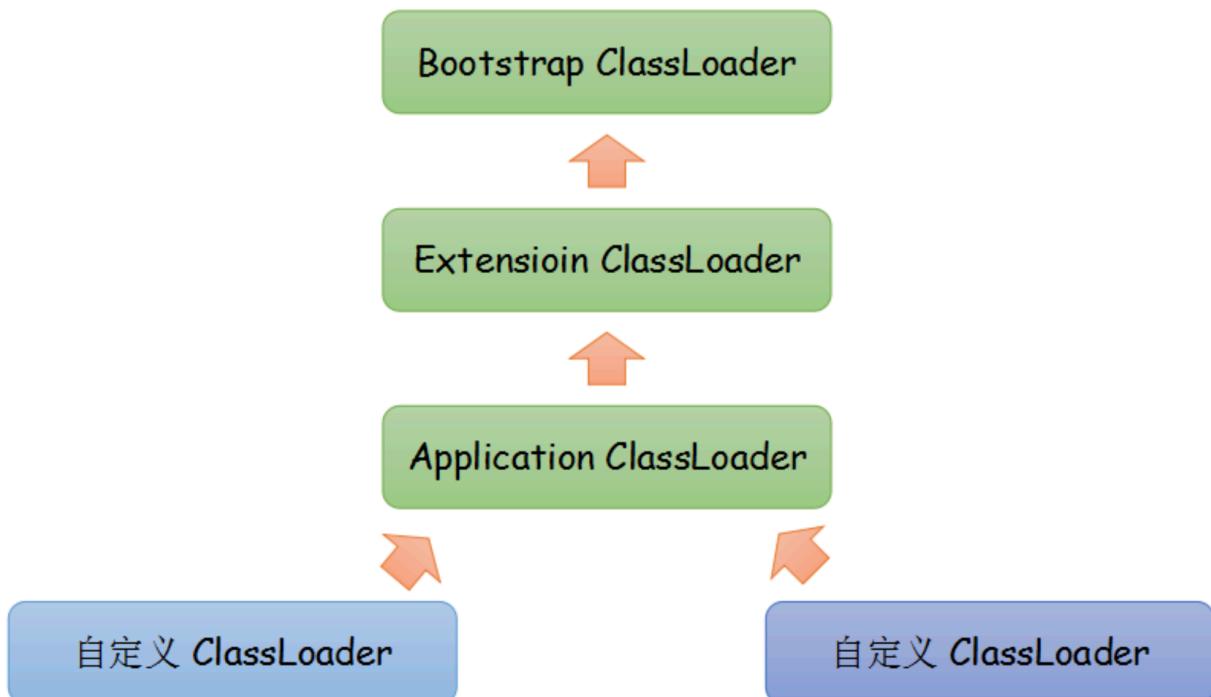
- 类初始化方法。编译器会按照其出现顺序，收集类变量的赋值语句、静态代码块，最终组成类初始化方法。**类初始化方法一般在类初始化的时候执行。**
- 对象初始化方法。编译器会按照其出现顺序，收集成员变量的赋值语句、普通代码块，最后收集构造函数的代码，最终组成对象初始化方法。**对象初始化方法一般在实例化类对象的时候执行。**

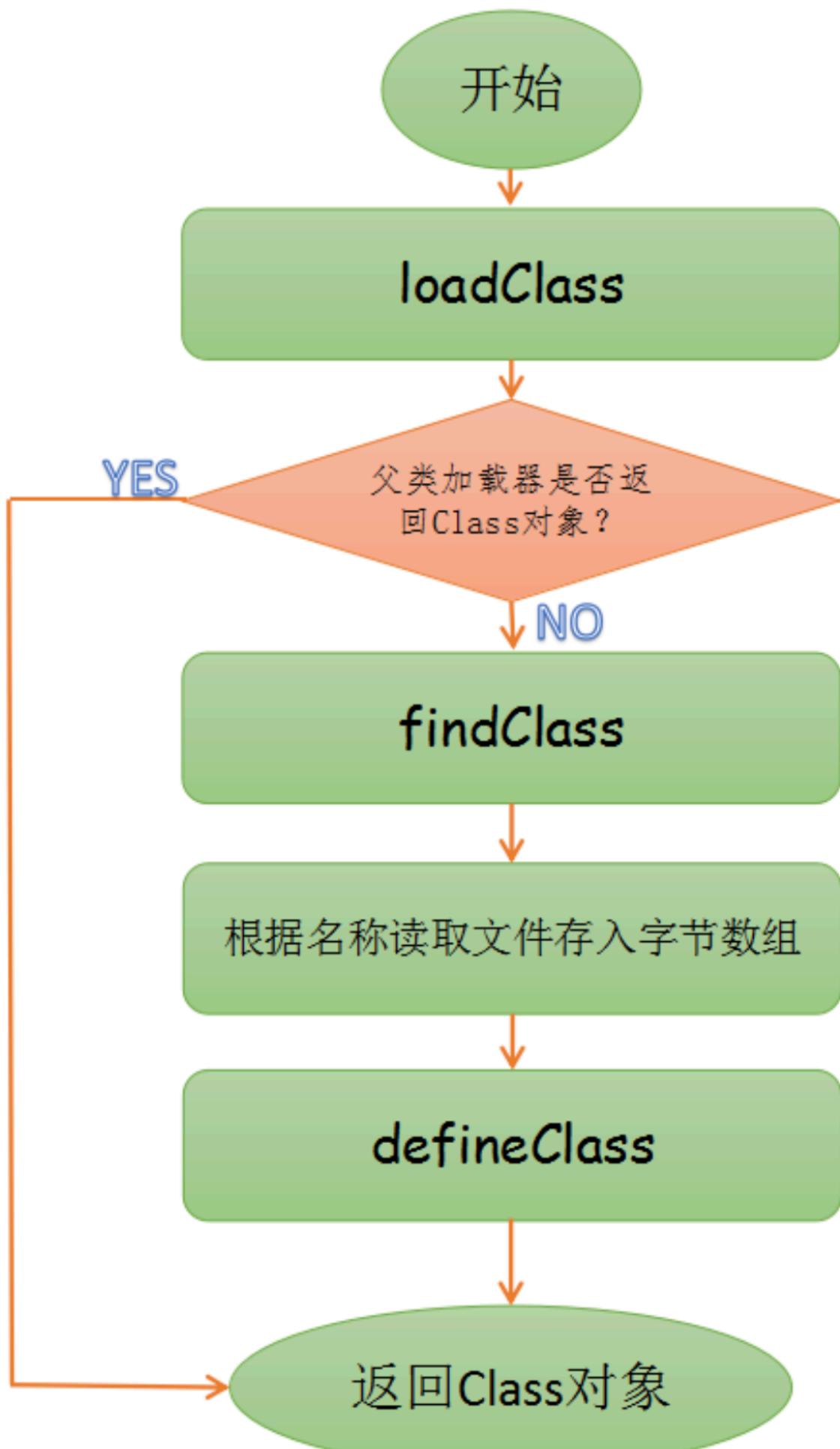
类加载过程解析：https://www.cnblogs.com/chanshuai/p/the_java_class_load_mechanism.html，结合_01Common Book类进行解析

1. 类的实例化顺序，比如父类静态数据，构造函数，字段，子类静态数据，构造函数，字段，当new的时候，他们的执行顺序

2. 双亲委派模型

- 类加载阶段分为加载、连接、初始化三个阶段，而加载阶段需要通过类的全限定名来获取定义了此类的二进制字节流。**Java特意把这一步抽出来用类加载器来实现。**把这一步骤抽离出来使得应用程序可以按需自定义类加载器。并且得益于类加载器，OSGI、热部署等领域才得以在JAVA中得到应用。
- 在Java中任意一个类都是由这个类本身和加载这个类的类加载器来确定这个类在JVM中的唯一性。也就是你用你A类加载器加载的 `com.aa.ClassA` 和你A类加载器加载的 `com.aa.ClassA` 它们是不同的，也就是用 `instanceof` 这种对比都是不同的。所以即使都来自于同一个class文件但是由不同类加载器加载的那就是两个独立的类。
- 如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即 `ClassNotFoundException`），子加载器才会尝试自己去加载。





1. 什么是双亲委派模型

说明双亲委派模型之前需要了解Java类加载机制，就是加载-验证-准备-解析-初始化-使用-卸载的过程

所谓双亲委派是指每次收到类加载请求时，先将请求委派给父类加载器完成（所有加载请求最终会委派到顶层的Bootstrap ClassLoader加载器中），如果父类加载器无法完成这个加载（该加载器的搜索范围中没有找到对应的类），子类尝试自己加载，如果都没加载到，则会抛出 ClassNotFoundException 异常，看到这里其实就解释了文章开头提出的第一个问题，父加载器已经加载了JDK 中的 String.class 文件，所以我们不能定义同名的 String.java 文件。

2. 双亲委派模型的意义

简单来说就是为了代码安全：

因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要 ClassLoader 再加载一次。考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的String来动态替代java核心api中定义的类型，这样会存在非常大的安全隐患，而双亲委托的方式，就可以避免这种情况，因为String已经在启动时就被引导类加载器（Bootstrap ClassLoader）加载，所以用户自定义的ClassLoader永远也无法加载一个自己写的String，除非你改变 JDK 中 ClassLoader 搜索类的默认算法。

Ref: <https://juejin.im/post/5d27dc7de51d4510a37bac85>

3. 类加载器

说出你知道的类加载器

类加载器负责加载所有的类，其为所有被载入内存中的类生成一个java.lang.Class实例对象。一旦一个类被加载如JVM中，同一个类就不会被再次载入了。正如一个对象有一个唯一的标识一样，一个载入JVM的类也有一个唯一的标识。在Java中，一个类用其全限定类名（包括包名和类名）作为标识；但在JVM中，一个类用其全限定类名和其类加载器作为其唯一标识。例如，如果在pg的包中有一个名为Person的类，被类加载器ClassLoader的实例kl负责加载，则该Person类对应的Class对象在JVM中表示为(Person.pg.kl)。这意味着两个类加载器加载的同名类：(Person.pg.kl) 和 (Person.pg.kl2) 是不同的、它们所加载的类也是完全不同、互不兼容的。

首先，先要知道什么是类加载器。简单说，类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，转为 class 对象。如果站在 JVM 的角度来看，只存在两种类加载器：

- **启动类加载器（Bootstrap ClassLoader）**：由C++语言实现（针对HotSpot），负责将存放在<JAVA_HOME>\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以 不允许直接通过引用进行操作。
- 其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：
 - **扩展类加载器（Extension ClassLoader）**：负责加载<JAVA_HOME>\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库。
 - **系统类加载器（System ClassLoader/Application ClassLoader）**：负责加载用户类路径(classpath)上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。如果没有特别指定，则用户自定义的类加载器都以此类加载器作为父加载器。由Java语言实现，父类加载器为ExtClassLoader。

类加载器加载Class大致要经过如下8个步骤：

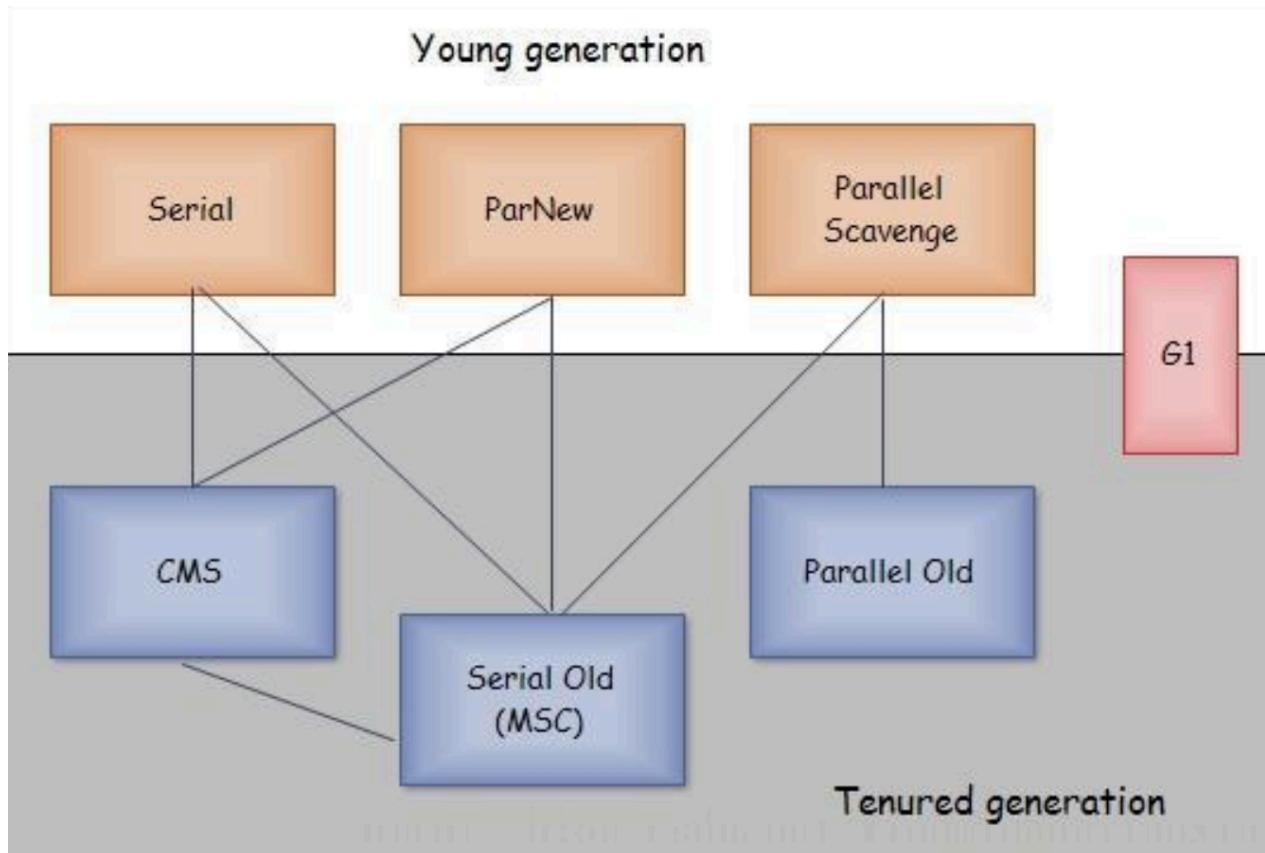
1. 检测此Class是否载入过，即在缓冲区中是否有此Class，如果有直接进入第8步，否则进入第2步。
2. 如果没有父类加载器，则要么Parent是根类加载器，要么本身就是根类加载器，则跳到第4步，如果父类加载器存在，则进入第3步。
3. 请求使用父类加载器去载入目标类，如果载入成功则跳至第8步，否则接着执行第5步。
4. 请求使用根类加载器去载入目标类，如果载入成功则跳至第8步，否则跳至第7步。
5. 当前类加载器尝试寻找Class文件，如果找到则执行第6步，如果找不到则执行第7步。
6. 从文件中载入Class，成功后跳至第8步。
7. 抛出ClassNotFoundException异常。
8. 返回对应的java.lang.Class对象。

1. 如何自定义类加载器

2. 如何破坏双亲委派模型

12. GC垃圾回收器 ★★★★★★

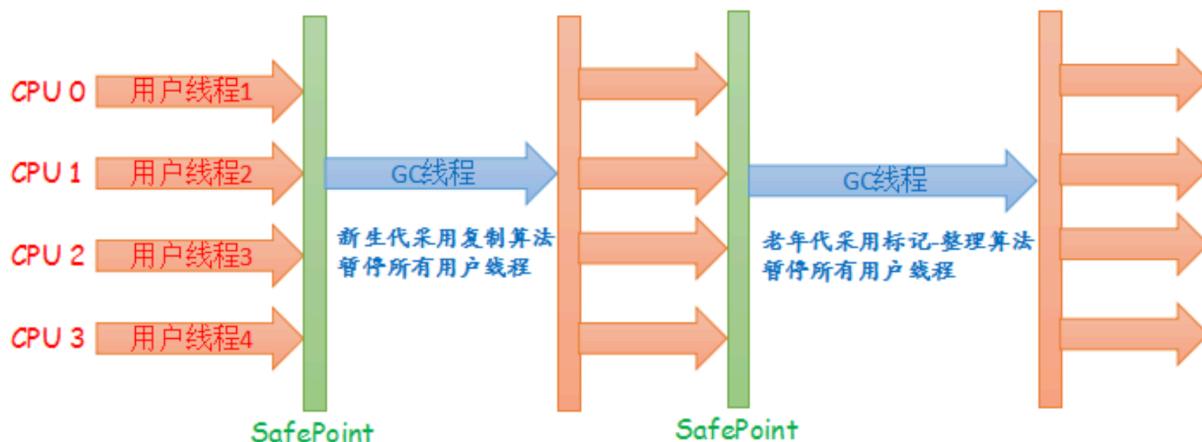
1. 总览



图中展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。

- Safe Point
 - 安全点顾名思义是指一些特定的位置，当线程运行到这些位置时，线程的一些状态可以被确定(the thread's representation of its Java machine state is well described)，比如记录OopMap的状态，从而确定GC Root的信息，使JVM可以安全的进行一些操作，比如开始GC。
 - 位置
 - 循环的末尾
 - 方法返回前
 - 调用方法之后
 - 抛出异常的位置
- Stop The World
 - 不管选择哪种GC算法，stop-the-world都是不可避免的。Stop-the-world意味着从应用中停下来并进入到GC执行过程中去。一旦Stop-the-world发生，除了GC所需的线程外，其他线程都将停止工作，中断了的线程直到GC任务结束才继续它们的任务。**GC调优通常就是为了改善stop-the-world的时间。**
- <https://segmentfault.com/a/1190000004233812>

2. Serial收集器



特性：

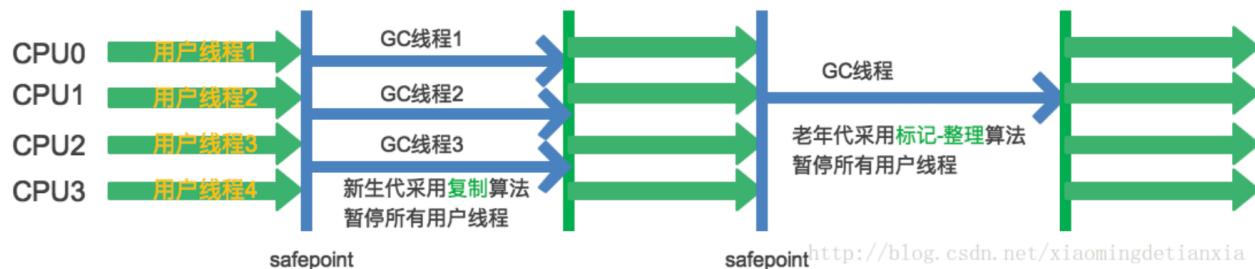
- 这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束（**Stop The World**）。
- 只有一个GC线程处理任务

应用场景：Serial收集器是虚拟机运行在Client模式下的默认新生代收集器。

优势：

- 简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。
- 其中**Serial GC务必不要在生产环境的服务器上使用**，这种GC是为单核CPU上的桌面应用设计的。使用Serial GC会明显的损耗应用的性能。

3. ParNew回收器



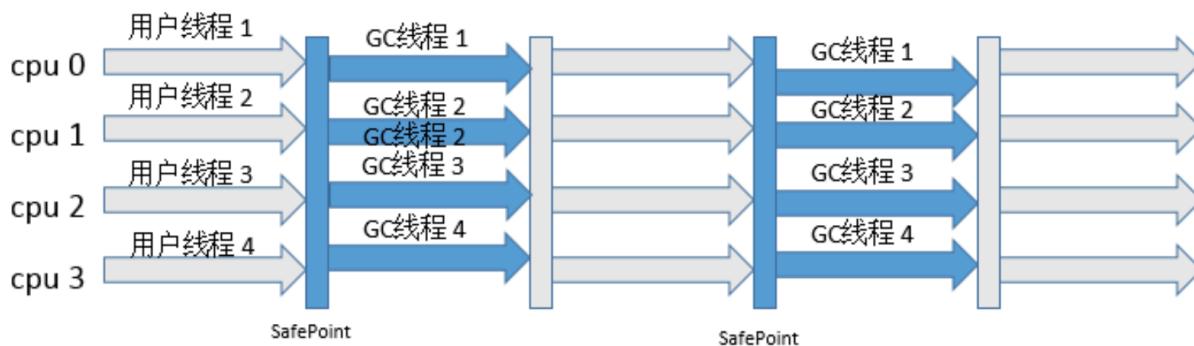
特性：ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。

应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器。

很重要的原因是：除了Serial收集器外，目前只有它能与CMS收集器配合工作。在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器，这款收集器是HotSpot虚拟机中第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。不幸的是，CMS作为老年代的收集器，却无法与JDK 1.4.0中已经存在的新生代收集器Parallel Scavenge配合工作，所以在JDK 1.5中使用CMS来收集老年代的时候，新生代只能选择ParNew或者Serial收集器中的一个。

Serial收集器 VS ParNew收集器：ParNew收集器在单CPU的环境中绝对不会比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。然而，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。

4. Parallel Scavenge回收器



Parallel Scavenge/Parallel Old 收集器运行示意图 <http://blog.csdn.net/xiaomingdetianxia>

特性：Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。

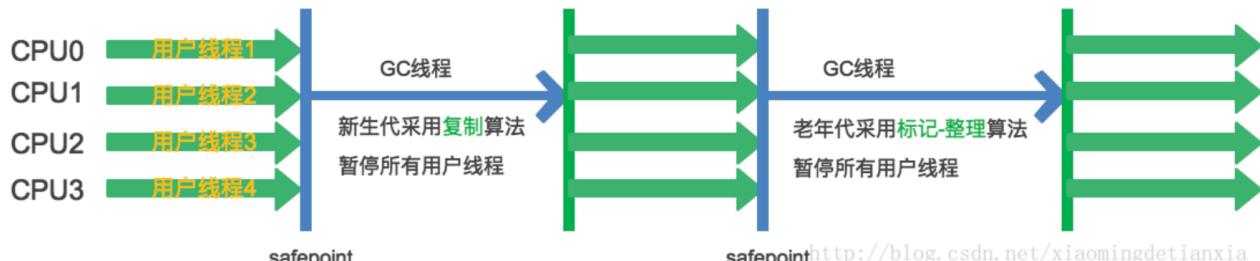
应用场景：停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

对比分析：Parallel Scavenge收集器 VS CMS等收集器：Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。

Parallel Scavenge收集器 VS ParNew收集器： Parallel Scavenge收集器与ParNew收集器的一个重要区别是它具有自适应调节策略。

GC自适应的调节策略： Parallel Scavenge收集器有一个参数-XX:+UseAdaptiveSizePolicy。当这个参数打开之后，就不需要手工指定新生代的大小、Eden与Survivor区的比例、晋升老年代对象年龄等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

5. Serial Old回收器



特性： Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记－整理算法。

应用场景： Client模式 Serial Old收集器的主要意义也是在于给Client模式下的虚拟机使用。

Server模式 如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用，另一种用途就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

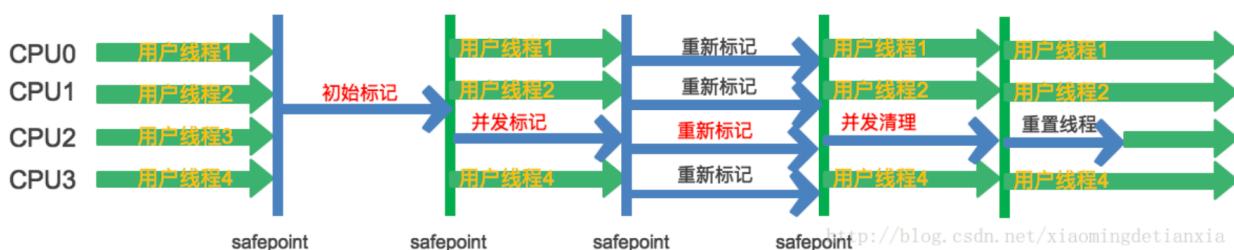
6. Parallel Old回收器

特性： Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。

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

这个收集器是在JDK 1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old收集器外别无选择（Parallel Scavenge收集器无法与CMS收集器配合工作）。由于老年代Serial Old收集器在服务端应用性能上的“拖累”，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合“给力”。直到Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合。

7. CMS回收器 ★



特性：

- CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为 目标 的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。
- 清理的过程和程序运行是并发的，不会暂停其他工作线程。
- CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：

初始标记（CMS initial mark） 初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，需要“Stop The World”。

并发标记（CMS concurrent mark） 并发标记阶段就是进行GC Roots Tracing的过程。

重新标记（CMS remark） 重新标记阶段是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短，仍然需要“Stop The World”。

并发清除（CMS concurrent sweep） 并发清除阶段会清除对象。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

优点： CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿。

缺点： CMS收集器对CPU资源非常敏感 其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。CMS默认启动的回收线程数是 $(CPU\text{数量}+3)/4$ ，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大。

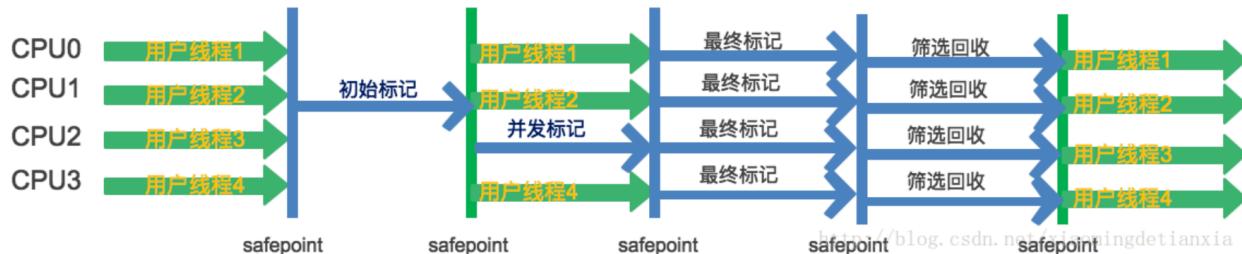
CMS收集器无法处理浮动垃圾 CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。

由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

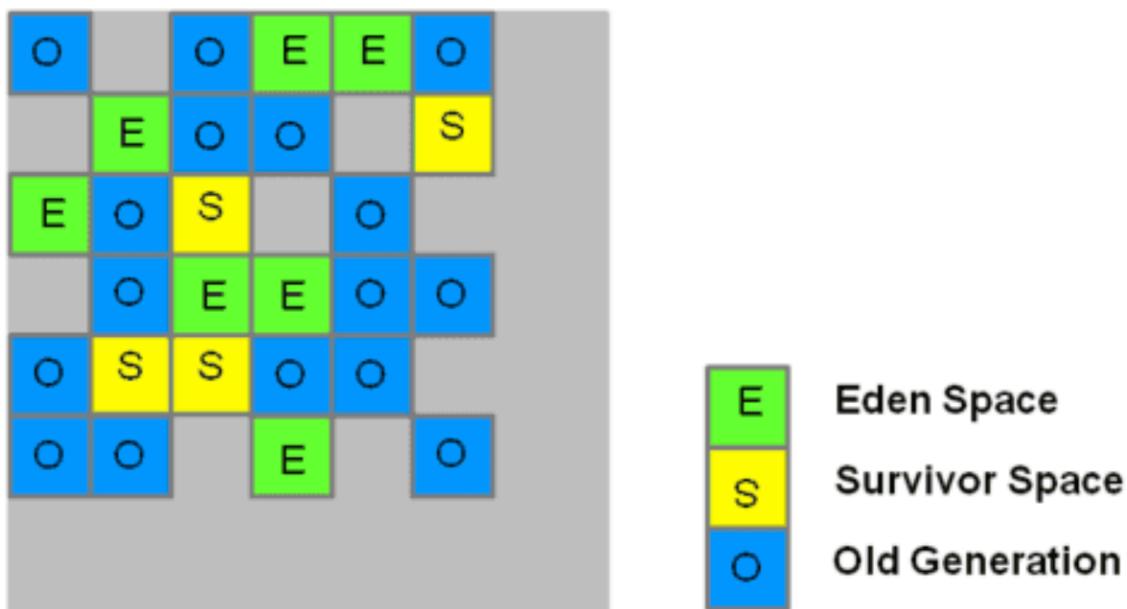
CMS收集器会产生大量空间碎片 CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。

空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

7. G1收集器



G1 Heap Allocation



特性： G1 (Garbage-First) 是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点。

并行与并发 G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。

分代收集 与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整合 与CMS的“标记—清理”算法不同，G1从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。

可预测的停顿 这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也就是Garbage-First名称的来由）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

执行过程： G1收集器的运作大致可划分为以下几个步骤：

初始标记（Initial Marking） 初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这阶段需要停顿线程，但耗时很短。

并发标记（Concurrent Marking） 并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

最终标记（Final Marking） 最终标记阶段是为了修正正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这阶段需要停顿线程，但是可并行执行。

筛选回收（Live Data Counting and Evacuation） 筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。

8. G1和CMS区别

- CMS
 - 首先是支持并发清理，线程运行的时候依旧可以进行垃圾回收
 - 所以不能清理浮动垃圾【缺点】
 - 追求低停顿时间
 - 垃圾回收算法是Mark-Sweep，会产生大量的空间碎片
- G1
 - 使用Mark-Compact算法，不会产生内存碎片
 - 可以预测停顿时间

13. jstack、jmap、jstat

Jstack是Jdk自带的线程跟踪工具，用于打印指定Java进程的线程堆栈信息。

14. OOM

为什么会没有内存了呢？原因不外乎有两点：

- 1) 分配的少了：比如虚拟机本身可使用的内存（一般通过启动时的VM参数指定）太少。
 - 2) 应用用的太多，并且用完没释放，浪费了。此时就会造成内存泄露或者内存溢出。
 - 内存泄露：申请使用完的内存没有释放，导致虚拟机不能再次使用该内存，此时这段内存就泄露了，因为申请者不用了，而又不能被虚拟机分配给别人用。
 - 内存溢出：申请的内存超出了JVM能提供的内存大小，此时称之为溢出。
- 3) 最常见的OOM情况有以下三种：
 - java.lang.OutOfMemoryError: Java heap space ----->java堆内存溢出，此种情况最常见，一般由于内存泄露或者堆的大小设置不当引起。对于内存泄露，需要通过内存监控软件查找程序中的泄露代码，而堆大小可以通过虚拟机参数-Xms,-Xmx等修改。
 - java.lang.OutOfMemoryError: PermGen space ----->java永久代溢出，即方法区溢出了，一般出现于大量Class或者jsp页面，或者采用cglib等反射机制的情况，因为上述情况会产生大量的Class信息存储于方法区。此种情况可以通过更改方法区的大小来解决，使用类似-XX:PermSize=64m -XX:MaxPermSize=256m的形式修改。另外，过多的常量尤其是字符串也会导致方法区溢出。
 - java.lang.StackOverflowError -----> 不会抛OOM error，但也是比较常见的Java内存溢出。JAVA虚拟机栈溢出，一般是由于程序中存在死循环或者深度递归调用造成的，栈大小设置太小也会出现此种溢出。可以通过虚拟机参数-Xss来设置栈的大小。

15. JVM new一个对象具体发生了什么

在Java中我们创建对象都会用new进行创建，下面我来接收一下new之后对象创建及内存分配的具体的过程

一：虚拟机遇到一条new指令后，先去检查这条指令参数是否能在常量池中定位到一个类的符号引用，并检查这个符号引用代表的类是否已被加载、解析和初始化过，如果没有，那必须先执行相应的类加载过程。

二：类加载检查通过后，接下来虚拟机为新生对象分配内存，因为对象所需内存的大小在类加载后是完全确定的(用引用，堆中存放实例)，所以只需分配一个固定大小的内存即可。分配内存的方法用两种(JAVA虚拟机中)

①指针碰撞：假设Java堆中内存是绝对规整的，所有用过的内存都存放在一边，空闲的内存存在在另一边，中间放着一个以指针为分界点的指示器，分配内存就是把指针往空闲的那侧移动对象需要的内存即可。但是如果Java内存堆不是完整的，就没有办法进行简单的指针碰撞，

②空闲列表：空闲列表就是虚拟机先对内存分块，并用一个表记录每个内存块是否使用的情况，为对象分配内存时只需更新列表上的记录即可，选择哪一种分配方式取决于对中的内存使用情况是否完整。

但是在为对象分配内存的同时，需要考虑其它线程是否也在这一时刻需要分配，因此需要考虑到线程的安全问题，通过以下两种方法进行解决：

①同步处理[乐观锁]：一种是对分配内存的空间动作进行同步处理--实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性

②本地线程分配缓存TLAB:另一种是把内存分配的动作按照线程在不同的空间之中进行，及每个线程预先分配一定的缓存。

TLAB的全称是**Thread Local Allocation Buffer**，即线程本地分配缓存区，这是一个线程专用的内存分配区域。:)

三：内存分配完成后，虚拟机将分配到的内存空间都初始化为零值(不包括对象头)，如果采用的是TLAB的方法分配，则这一步骤在分配之前完成。这一步操作保证了对象的实例字段在Java代码中可以不付初始值就直接使用，程序访问的值为false。这个解决了以前我在上Java课与老师争论在执行构造方法是否会先初始化，明显我对了，但是当时没有拿出有说服力的证据。

四：在上面的步骤完成后，从虚拟机的视角，一个新的对象已经产生了，但是从Java程序的视角来看，对象创建方法才刚刚开始，还要执行对象的init方法（构造方法），一个对象才完成创建

004 多线程

1. volatile关键词的作用。

被volatile修饰的共享变量，就会具有以下两个特性：

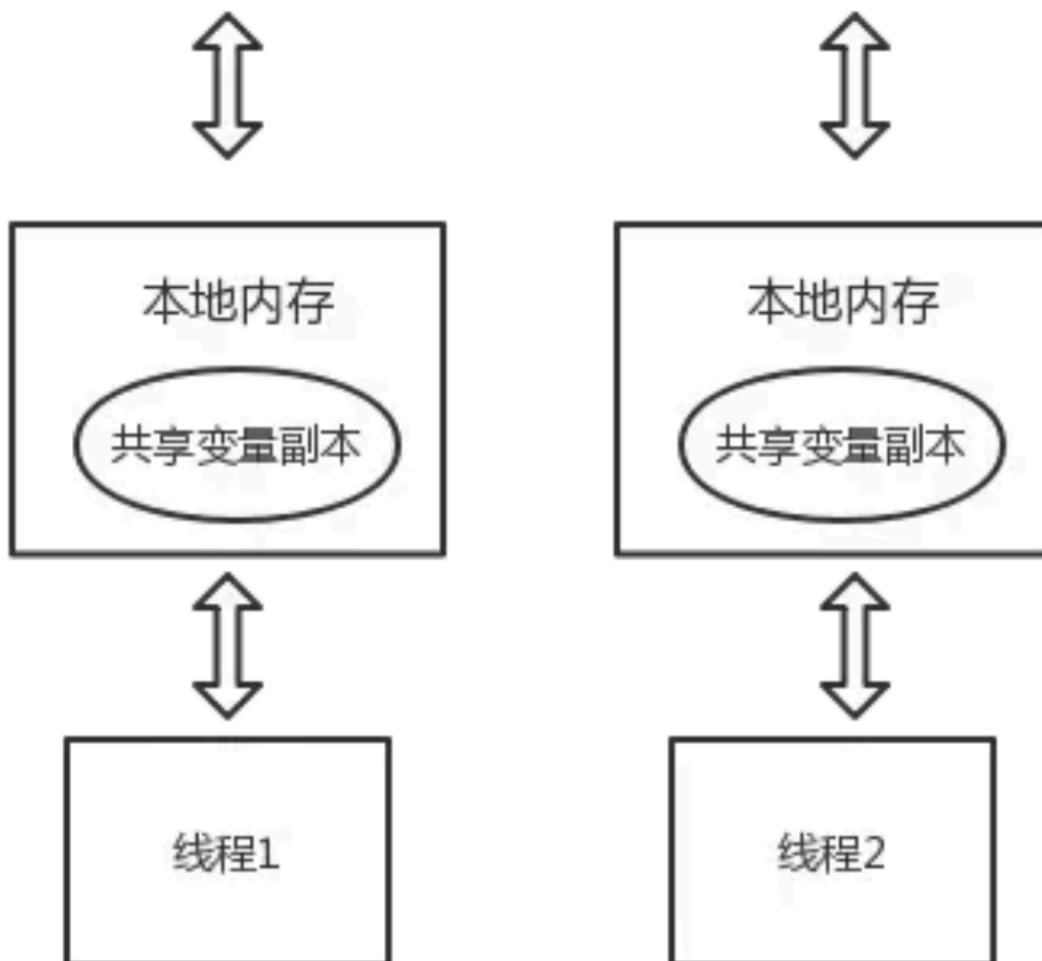
1. 保证了不同线程对该变量操作的内存可见性。
2. 禁止指令重排序。

1. volatile不保证原子性

2. volatile保证内存可见性

- 这个要是说起来可就多了，我就从Java内存模型开始说起吧。Java虚拟机规范试图定义一个Java内存模型(JMM)，以屏蔽所有类型的硬件和操作系统内存访问差异，让Java程序在不同的平台上能够达到一致的内存访问效果。简单地说，由于CPU执行指令的速度很快，但是内存访问速度很慢，差异不是一个量级，所以搞处理器的那群大佬们又在CPU里加了好几层高速缓存。
- 在Java内存模型中，对上述优化进行了一波抽象。JMM规定所有的变量都在主内存中，类似于上面提到的普通内存，每个线程又包含自己的工作内存，为了便于理解可以看成CPU上的寄存器或者高速缓存。因此，线程的操作都是以工作内存为主，它们只能访问自己的工作内存，并且在工作之前和之后，该值被同步回主内存。

主内存



3. **volatile**禁止指令重排序

2. 6种线程状态以及如何转换

1. 线程之间的状态是如何转换的

Java线程的生命周期中，存在几种状态。在Thread类里有一个枚举类型State，定义了线程的几种状态，分别有：

NEW：线程创建之后，但是还没有启动(not yet started)。这时候它的状态就是NEW

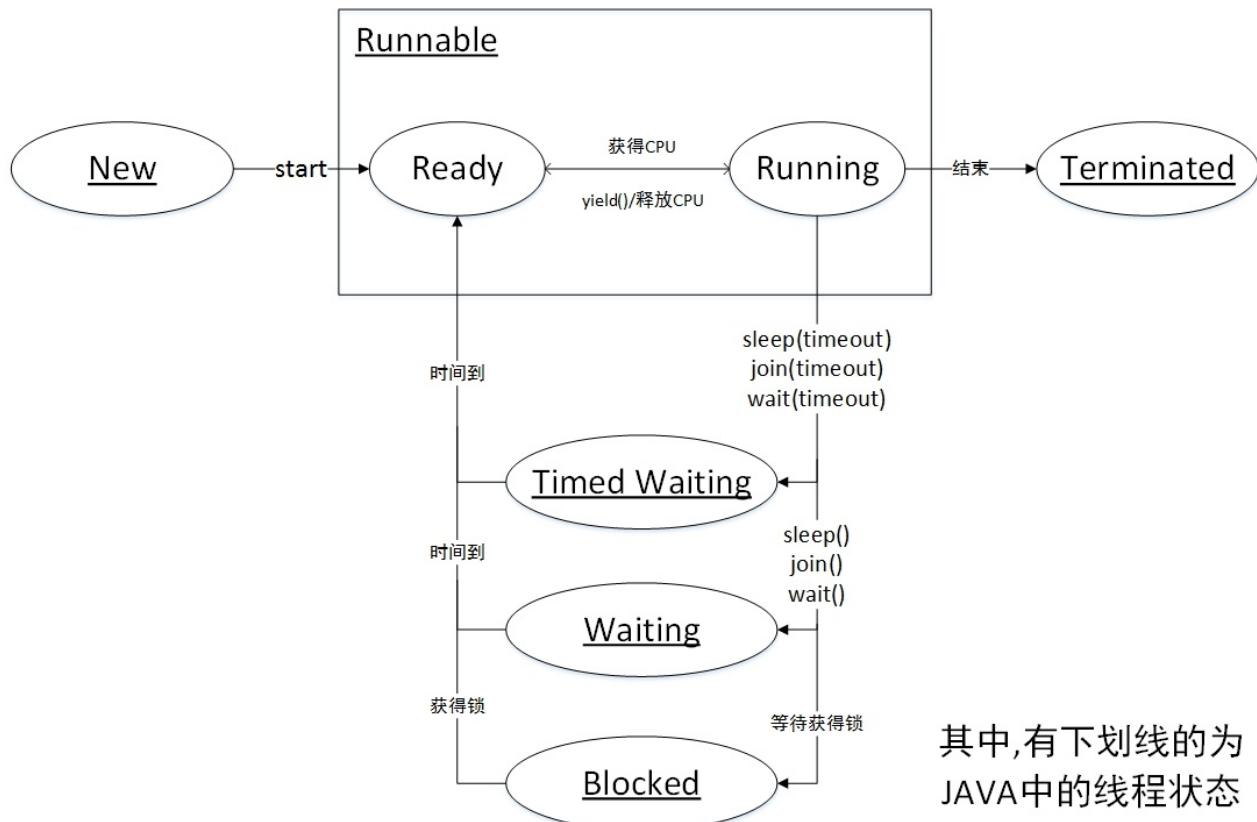
RUNNABLE：正在Java虚拟机下跑任务的线程的状态。在RUNNABLE状态下的线程可能会处于等待状态，因为它正在等待一些系统资源的释放，比如IO

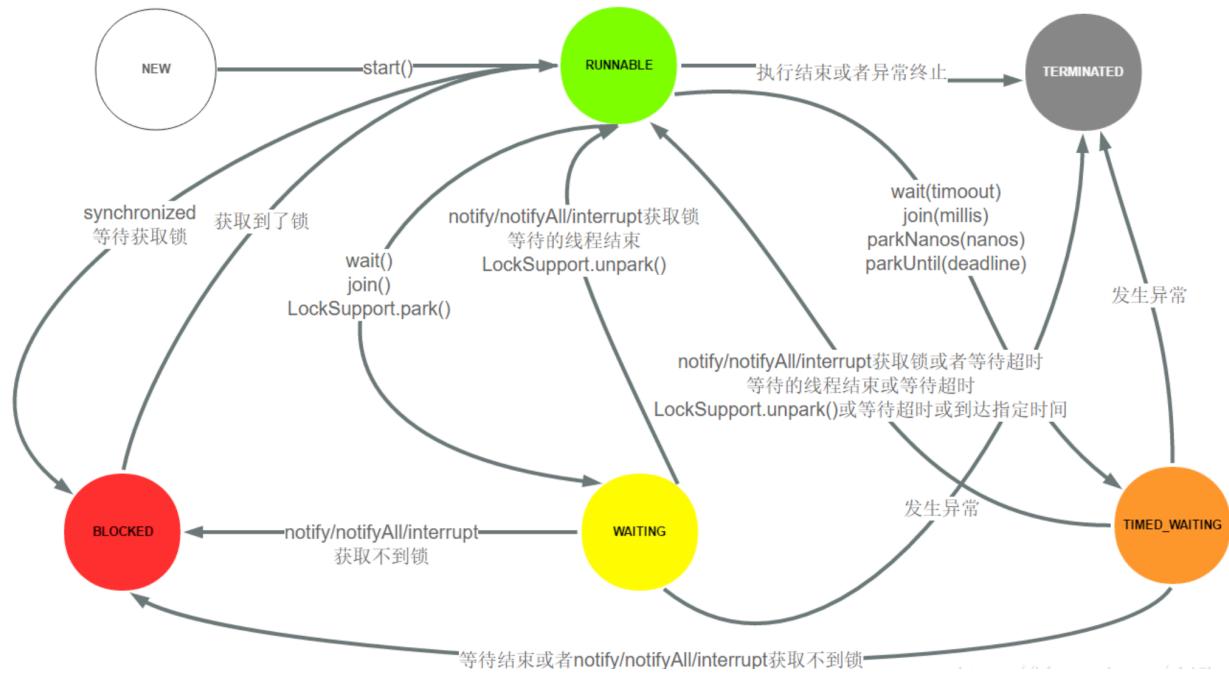
BLOCKED：阻塞状态，等待锁的释放，比如线程A进入了一个synchronized方法，线程B也想进入这个方法，但是这个方法的锁已经被线程A获取了，这个时候线程B就处于BLOCKED状态(比较常见)

WAITING：等待状态，处于等待状态的线程是由于执行了3个方法中的任意方法。1. Object的wait方法，并且没有使用timeout参数；2. Thread的join方法，没有使用timeout参数 3. LockSupport的park方法。处于waiting状态的线程会等待另外一个线程处理特殊的行为。再举个例子，如果一个线程调用了一个对象的wait方法，那么这个线程就会处于waiting状态直到另外一个线程调用这个对象的notify或者notifyAll方法后才会解除这个状态

TIMED_WAITING：有等待时间的等待状态，比如调用了以下几个方法中的任意方法，并且指定了等待时间，线程就会处于这个状态。1. Thread.sleep方法 2. Object的wait方法，带有时间 3. Thread.join方法，带有时间 4. LockSupport的parkNanos方法，带有时间 5. LockSupport的parkUntil方法，带有时间

TERMINATED：线程中止的状态，这个线程已经完整地执行了它的任务





2. wait和sleep区别

- sleep()不释放同步锁,wait()释放同步锁
- sleep()是Thread类特有的方法, wait()是Object类的方法
- sleep()可以在任何地方使用, wait()只能在synchronized同步语句中使用

3. run和start区别

run()是在当前线程中调用线程类的run()方法, start()是开启一个新的线程

3. 乐观锁和悲观锁

悲观锁: synchronized和ReentrantLock

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：版本控制机制和CAS算法

CAS

应用场景：JUC.atomic包

`compare and swap` (比较与交换) , 是一种有名的无锁算法。无锁编程, 即不使用锁的情况下实现多线程之间的变量同步, 也就是在没有线程被阻塞的情况下实现变量的同步, 所以也叫非阻塞同步 (Non-blocking Synchronization) 。CAS算法涉及到三个操作数

需要读写的内存值 `v`

进行比较的值 `A`

拟写入的新值 `B`

当且仅当 `v` 的值等于 `A`时, CAS通过原子方式用新值`B`来更新`v`的值, 否则不会执行任何操作 (比较和替换是一个原子操作)。一般情况下是一个自旋操作, 即不断的重试。

缺点:

CPU性能开销: 如果一个线程不断尝试修改值, 都没有成功, 计算造成的开销

ABA问题:

如果一个变量`v`初次读取的时候是`A`值, 并且在准备赋值的时候检查到它仍然是`A`值, 那我们就能说明它的值没有被其他线程修改过了吗? 很明显是不能的, 因为在这段时间它的值可能被改为其他值, 然后又改回`A`, 那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA" 问题。

synchronized和CAS乐观锁的比较: 单的来说**CAS**适用于写比较少的情况下 (多读场景, 冲突一般较少) , **synchronized**适用于写比较多的情况下 (多写场景, 冲突一般较多)

对于资源竞争较少 (线程冲突较轻) 的情况, 使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源; 而CAS基于硬件实现, 不需要进入内核, 不需要切换线程, 操作自旋几率较少, 因此可以获得更高的性能。

对于资源竞争严重 (线程冲突严重) 的情况, CAS自旋的概率会比较大, 从而浪费更多的CPU资源, 效率低于synchronized。

4. JUC并发包

1. synchronized

1. 使用场景

分类	具体分类	被锁的对象	伪代码
方法	实例方法	类的实例对象	//实例方法，锁住的是该类的实例对象 public synchronized void method() { }
	静态方法	类对象	//静态方法，锁住的是类对象 public static synchronized void method1() { }
代码块	实例对象	类的实例对象	//同步代码块，锁住的是该类的实例对象 synchronized (this) { }
	class对象	类对象	//同步代码块，锁住的是该类的类对象 synchronized (SynchronizedDemo.class) { }
	任意实例对象Object	实例对象Object	//同步代码块，锁住的是配置的实例对象 //String对象作为锁 String lock = ""; synchronized (lock) { }

2. monitor监视器

- 执行同步代码块后首先要先执行**monitorenter**指令，退出的时候**monitorexit**指令。通过分析之后可以看出，使用Synchronized进行同步，其关键就是必须要对对象的监视器monitor进行获取，当线程获取monitor后才能继续往下执行，否则就只能等待。而这个获取的过程是互斥的，即同一时刻只有一个线程能够获取到monitor。上面的demo中在执行完同步代码块之后紧接着再会去执行一个静态同步方法，而这个方法锁的对象依然就这个类对象，那么这个正在执行的线程还需要获取该锁吗？答案是不必的，从上图中就可以看出来，执行静态同步方法的时候就只有一条**monitorexit**指令，并没有**monitorenter**获取锁的指令。这就是**锁的重入性**，即在同一锁程中，线程不需要再次获取同一把锁。Synchronized先天具有重入性。每个对象拥有一个计数器，当线程获取该对象锁后，计数器就会加一，释放锁后就会将计数器减一。
- 每个对象都有自己的监视器，当这个对象由同步块或者这个对象的同步方法调用时，执行方法的线程必须先获取该对象的监视器才能进入同步块和同步方法，如果没有获取到监视器的线程将会被阻塞在同步块和同步方法的入口处，进入到BLOCKED状态。

参考：<https://www.jianshu.com/p/d53bf830fa09>

2. ReentrantLock

```
//创建一个非公平锁，默认是非公平锁
Lock lock = new ReentrantLock();
Lock lock = new ReentrantLock(false);

//创建一个公平锁，构造传参true
Lock lock = new ReentrantLock(true);
```

1. 公平锁实现的原理是什么

Sync

NonfairSync

2. 非公平锁的实现原理是什么

3. AbstractQueuedSynchronizer 【AQS】

源码分析：<https://www.javadoop.com/post/AbstractQueuedSynchronizer>

4. ConcurrentHashMap

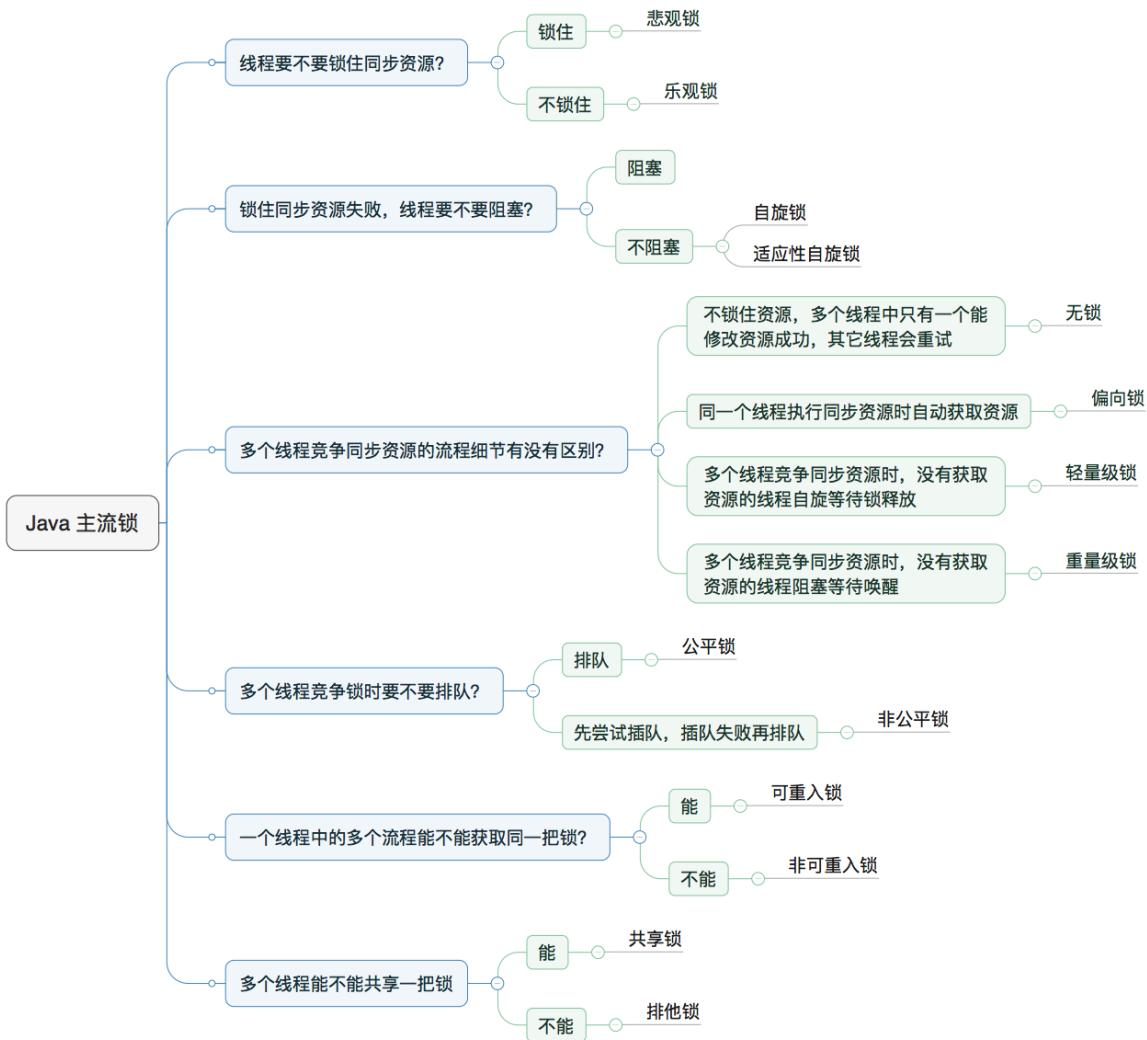
5. CopyOnWriteArrayList

5. Java里有哪些锁? ★★★★☆

美团技术文章：<https://tech.meituan.com/2018/11/15/java-lock.html>

- 按照特性有不同的划分规则，同一个锁可能会有多种特性，比如ReentrantLock是悲观锁、0非公平锁（默认）、可重入锁、独享锁。
- AQS的应用比较广泛

高层类	Lock	同步器	阻塞队列	Executor	并发容器
基础类		AQS	非阻塞数据结构	原子变量类	
	<code>volatile</code> 变量的读/写 CAS				

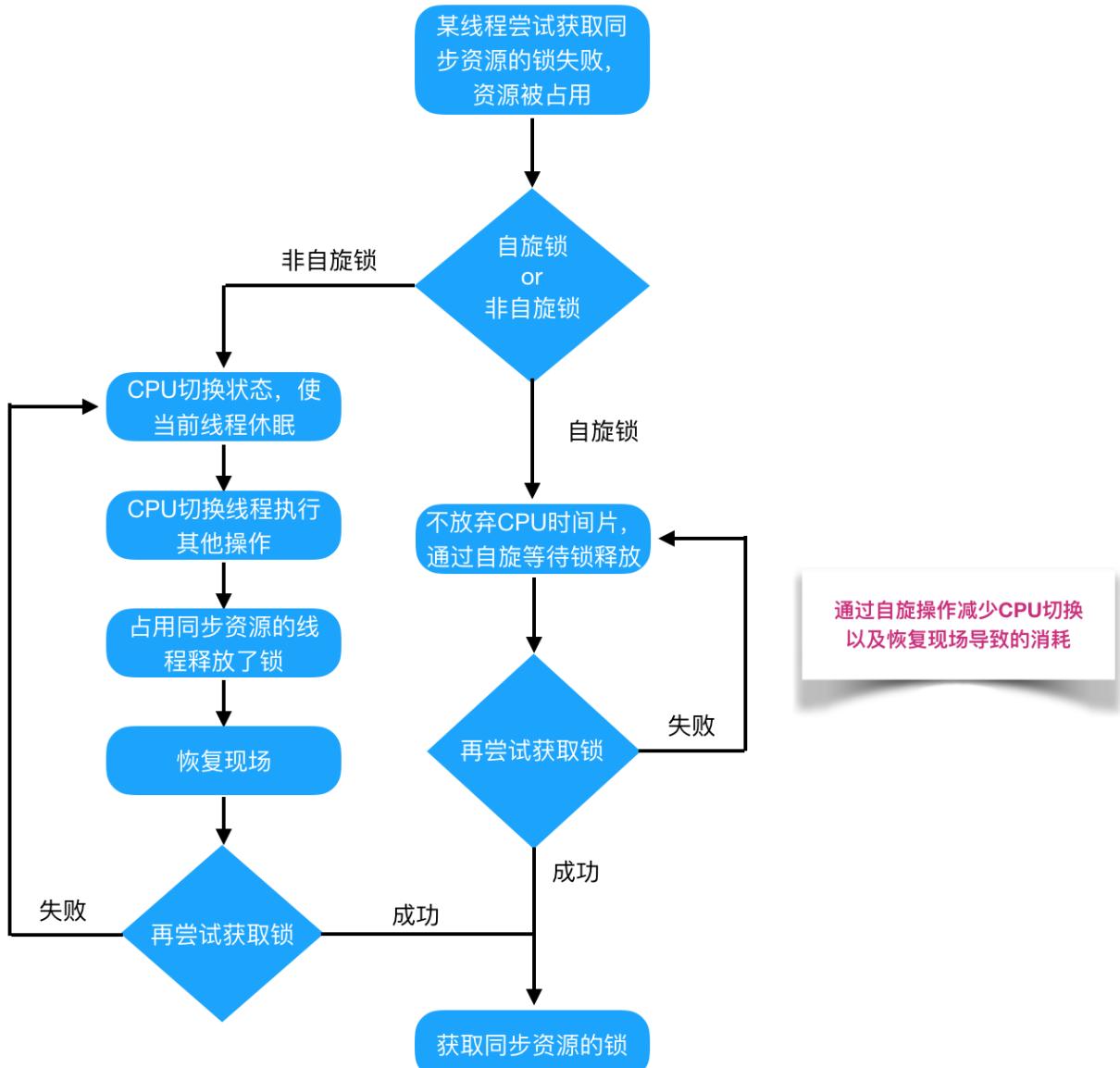


(1) 自旋锁:

阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。

在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。

而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。



应用场景：CAS (java.util.concurrent.atomicInteger) 重新尝试更新值的过程

(2) 公平锁&非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

应用场景：ReentrantLock默认是非公平锁，可以通过构造方法实现公平锁。

(3) 重入锁&非重入锁

可重入锁又名递归锁，是指同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。

```
public class Widget {  
    public synchronized void doSomething() {  
        System.out.println("方法1执行...");  
        doOthers();  
    }  
  
    public synchronized void doOthers() {  
        System.out.println("方法2执行...");  
    }  
}
```

(4) 独享锁&共享锁

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

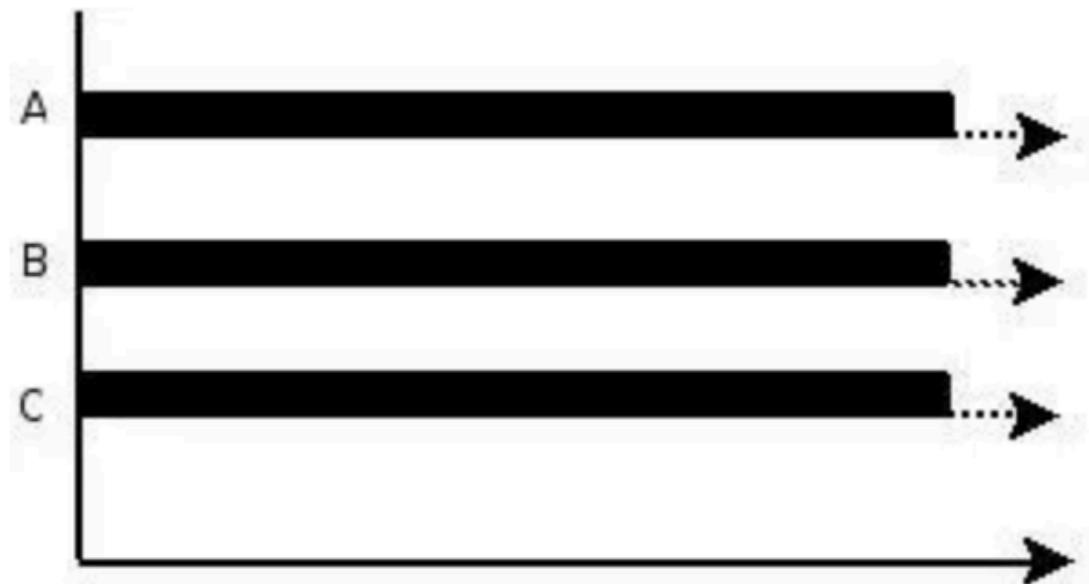
读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。

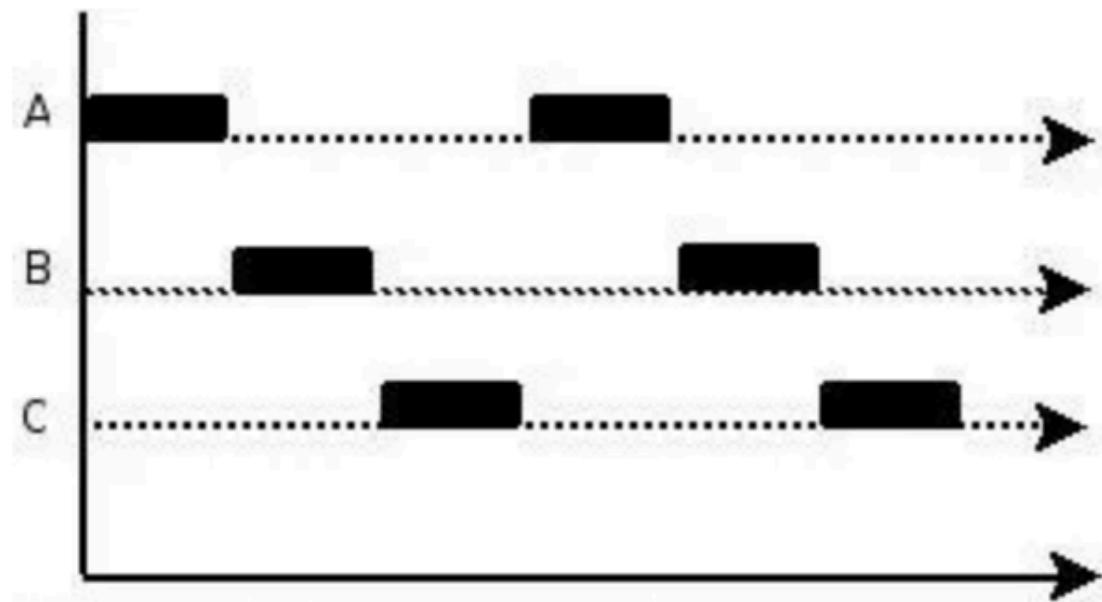
独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

6. 并发和并行

- 并发:一个处理器同时处理多个任务。【fake 同时】
- 并行:多个处理器或者是多核的处理器同时处理多个不同的任务 【true 同时】

前者是逻辑上的同时发生 (simultaneous)，而后者是物理上的同时发生。【是否使用多个处理器】





7. 无锁队列

在并发编程中，我们可能经常需要用到线程安全的队列，java为此提供了两种模式的队列：阻塞队列和非阻塞队列。

注：阻塞队列和非阻塞队列如何实现线程安全？

- 阻塞队列可以用一个锁（入队和出队共享一把锁）或者两个锁（入队使用一把锁，出队使用一把锁）来实现线程安全，JDK中典型的实现是 `BlockingQueue`；
- 非阻塞队列可以用循环CAS的方式来保证数据的一致性，来达到线程安全的目的。

8. Atomic包

创建多线程的几种姿势

1. Thread和Runnable的区别

- **Thread的局限性：**可以看到使用Thread是继承关系，而使用Runnable是实现关系。我们知道java不支持多继承，如果要实现多继承就得要用implements，所以使用上Runnable更加的灵活。
- **Runnable支持资源共享：**Runnable是可以共享数据的，多个Thread可以同时加载一个Runnable，当各自Thread获得CPU时间片的时候开始运行runnable，runnable里面的资源是被共享的。

2. 什么情况下使用Callable

Callable一般是配合线程池一起使用的，可以获取线程执行的返回结果

```

import java.util.concurrent.*;

public class _Callable {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        Future<Integer> result = executor.submit(task);
        executor.shutdown();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        System.out.println("主线程在执行任务");

        try {
            System.out.println("task运行结果"+result.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("所有任务执行完毕");
    }
}

class Task implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        System.out.println("子线程在进行计算");
        Thread.sleep(3000);
        int sum = 0;
        for(int i=1;i<101;i++)
            sum += i;
        return sum;
    }
}

```

3. Future/FutureTask

可以看出RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

使用场景：利用FutureTask和ExecutorService，可以用多线程的方式提交计算任务，主线程继续执行其他任务，当主线程需要子线程的计算结果时，在异步获取子线程的执行结果。

```

import java.util.concurrent.*;

```

```
public class _FutureTask {
    public static void main(String[] args) {
        //第一种方式
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        FutureTask<Integer> futureTask = new FutureTask<Integer>(task);
        executor.submit(futureTask);
        executor.shutdown();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        System.out.println("主线程在执行任务");

        try {
            System.out.println("task运行结果"+futureTask.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("所有任务执行完毕");
    }
}
```

9. synchronized实现原理

005 JDK8

函数式编程：

我们最常用的面向对象编程（Java）属于命令式编程（Imperative Programming）这种编程范式。常见的编程范式还有逻辑式编程（Logic Programming），函数式编程（Functional Programming）。

函数式编程作为一种编程范式，在科学领域，是一种编写计算机程序数据结构和元素的方式，它把计算过程当做是数学函数的求值，而避免更改状态和可变数据。

- 函数式编程带来的好处尤为明显。这种代码更多地表达了业务逻辑的意图，而不是它的实现机制。易读的代码也易于维护、更可靠、更不容易出错。【类似于框架的作用】

1. lambda表达式★★★★★

- 为什么要使用lambda表达式
 - 代码紧凑、简洁
- 使用Java 8 Lambda表达式可以实现更高的效率。通过使用具有多核的CPU，用户可以通过使用lambda并行处理集合来利用多核CPU。？？？

```
// 关于lambda表达式的语法
// ' ->' 是关键，左边是函数参数，右边是函数语句
//1. 遍历List
Arrays.asList("a", "v", "e").forEach(e -> {
    System.out.println(e);
    System.out.println("---");
});

//2. 创建线程
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!"))
.start();
```

2. 函数式接口★★

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。

```
@FunctionalInterface
interface GreetingService{
    //只能有这一个抽象方法
    void sayMessage(String message);
}

public class _funcitonalInterface {

    public static void main(String[] args) {
        //代替匿名内部类
        GreetingService greetingService = message -> {
            System.out.println("get msg :" + message);
        };

        greetingService.sayMessage("gee");
    }
}
```

Consumer

Function

3. Stream流操作 ★★★★☆

流支持并行操作，而迭代器，for循环都是串行操作，所以流在多核处理上有强大优势。

主要是对于集合类的操作：stream是对集合对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作，或者大批量数据操作。

应用场景：只要给出需要对其包含的元素执行什么操作，比如“过滤掉长度大于10的字符串”、“获取每个字符串的首字母”等，Stream会隐式地在内部进行遍历，做出相应的数据转换。

```
List<Person> collect = res
    .stream()
    .filter(person -> {
        if ("男".equals(person.getSex()) && person.getAge() < 22)
            return true;
        return false;
    })
    .collect(Collectors.toList());

System.out.println(collect);
```

<https://www.jianshu.com/p/9fe8632d0bc2> 关于Stream的实际应用

主要是Filter和Map的使用

<https://blog.csdn.net/GoGleTech/article/details/79454151>

0x2 设计模式

1. 单例模式

1. 双重校验锁

双重校验是指在获得锁之后也需要重新校验，而不是获得两次锁

第一次判断singleton是否为null

第一次判断是在Synchronized同步代码块外进行判断，由于单例模式只会创建一个实例，并通过getInstance方法返回singleton对象，所以，第一次判断，是为了在singleton对象已经创建的情况下，避免进入同步代码块，提升效率。

第二次判断singleton是否为null

第二次判断是为了避免以下情况的发生。

(1)假设：线程A已经经过第一次判断，判断singleton=null，准备进入同步代码块。

(2)此时线程B获得时间片，犹豫线程A并没有创建实例，所以，判断singleton仍然=null，所以线程B创建了实例singleton。

(3)此时，线程A再次获得时间片，犹豫刚刚经过第一次判断singleton=null(不会重复判断)，进入同步代码块，这个时候，我们如果不加入第二次判断的话，那么线程A又会创造一个实例singleton，就不满足我们的单例模式的要求，所以第二次判断是很有必要的。

```
package _00_Java_language._design_pattern.single;

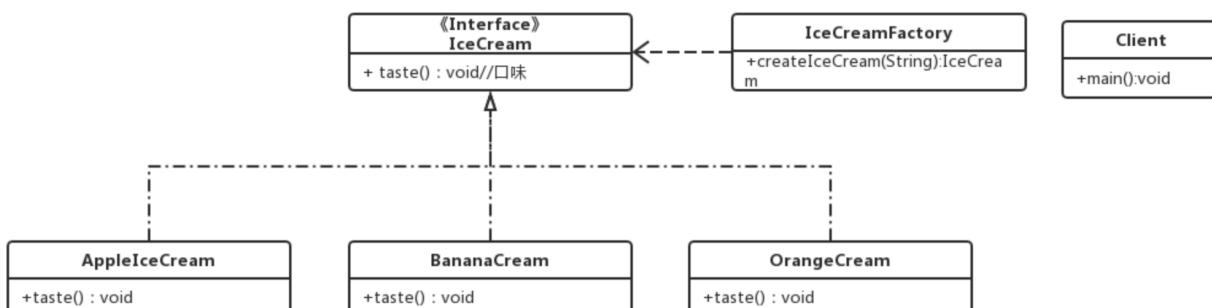
public class Singleton {
    private static volatile Singleton singleton;
    private Singleton(){}
    public static Singleton getSingleton(){
        if(singleton == null){
            synchronized (Singleton.class){
                if(singleton == null){
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

应用

数据库连接池

2. 工厂模式

1. 简单工厂模式



1. 创建统一的接口

```
public interface IceCream {
    public void taste();
}
```

2. 不同的实现类

```
public class AppleIceCream implements IceCream {  
  
    public void taste(){  
        System.out.println("这是苹果口味的冰激凌");  
    }  
}  
  
public class BananaIceCream implements IceCream {  
  
    public void taste() {  
        System.out.println("这是香蕉口味的冰激凌");  
    }  
}  
  
public class OrangeIceCream implements IceCream{  
  
    public void taste(){  
        System.out.println("这是橘子口味的冰激凌");  
    }  
}
```

3. 创建工厂

```
public class IceCreamFactory {  
  
    public static IceCream creamIceCream(String taste){  
  
        IceCream iceCream = null; //这里是关键-为什么要继承自同一接口  
  
        // 这里我们通过switch来判断，具体制作哪一种口味的冰激凌  
        switch(taste){  
  
            case "Apple":  
                iceCream = new AppleIceCream();  
                break;  
  
            case "Orange":  
                iceCream = new OrangeIceCream();  
                break;  
  
            case "Banana":  
                iceCream = new BananaIceCream();  
                break;  
  
            default:  
                break;  
        }  
    }  
}
```

```
    return iceCream;
}
}
```

4. 客户端调用

```
// 通过统一的工厂，传入不同参数调用生产冰激凌的方法去生产不同口味的冰激凌
public class Client {

    public static void main(String[] args) {

        IceCream appleIceCream = IceCreamFactory.creamIceCream("Apple");
        appleIceCream.taste();

        IceCream bananaIceCream = IceCreamFactory.creamIceCream("Banana");
        bananaIceCream.taste();

        IceCream orangeIceCream = IceCreamFactory.creamIceCream("Orange");
        orangeIceCream.taste();
    }
}
```

2. 工厂方法模式

3. 抽象工厂模式

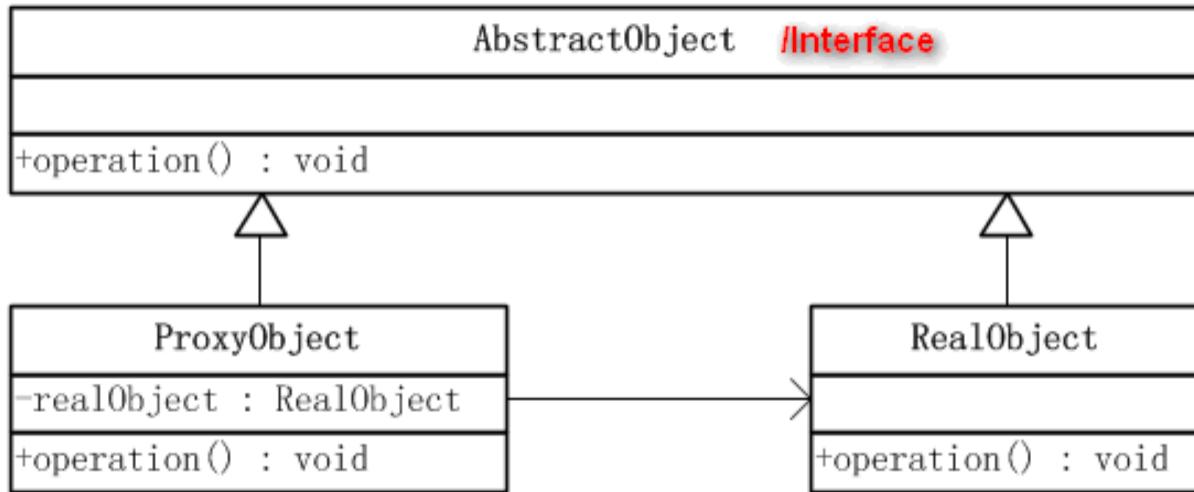
应用

线程池

3. 代理模式

1. 静态代理

- 静态代理的意义
 - 如果不能直接修改目标类但是又想要修改目标类中的方法的实现，这时候可以使用代理类增强目标类。



```

public interface Subject
{
    void doSomething();
}

public class RealSubject implements Subject {

    @Override
    public void doSomething()
    {
        System.out.println("做些什么呢？");
    }

}

public class ProxySubject implements Subject {

    private RealSubject realSubject = new RealSubject();

    @Override
    public void doSomething()
    {
        System.out.println("befor do something");
        realSubject.doSomething();
        System.out.println("after do something");
    }

}

```

4. 装饰器模式

应用：动态代理和静态代理

0x3 操作系统

1. 进程和线程

1. 进程和线程的区别

(1)CPU资源开销

(2)通信方式

(3)是否有独立的地址空间

进程是资源分配的最小单位，线程是程序执行的最小单位。

进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC）进行。不过如何处理好同步与互斥是编写多线程程序的难点。

但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

3. 进程

1. PCB (Process Control Block)

- 进程标识符信息。进程标识符用于唯一地标识一个进程。一个进程，通常有以下两个标识符：外部标识符，内部标识符。
- 处理机状态信息。处理机状态信息主要是由处理机各种寄存器中的内容所组成。
- 进程调度信息。在PCB中还存放了一些与进程调度和进程对换有关的信息，包括：进程状态、进程优先级、进程调度所需要的其他信息、事件。
- 进程控制信息。进程控制信息包括：程序和数据的地址、进程同步和通信机制、资源清单、链接指针。

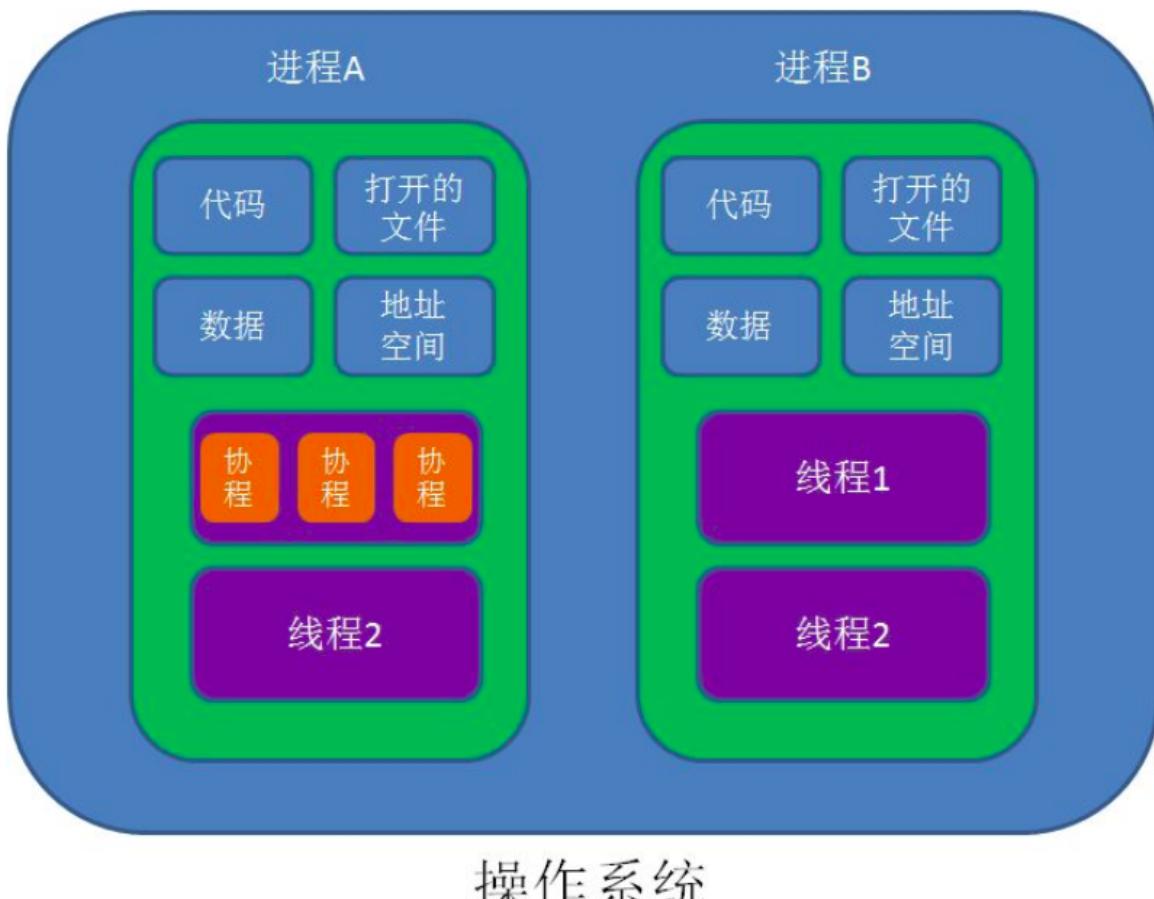
2. 协程是什么

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。

最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。

由于Java的原生语法中并没有实现协程。



2. 什么是操作系统的用户态，什么是操作系统的内核态？之间如何转换★★★★★

(1)为什么要区分用户态和内核态？

在计算机系统中，通常运行着两类程序：系统程序和应用程序，为了保证系统程序不被应用程序有意或无意地破坏，为计算机设置了两种状态：

系统态(也称为管态或核心态)，操作系统在系统态运行—运行操作系统程序

用户态(也称为目态)，应用程序只能在用户态运行—运行用户程序

在实际运行过程中，处理机会在系统态和用户态间切换。相应地，现代多数操作系统将 CPU 的指令集分为特权指令和非特权指令两类。

1) 特权指令—在系统态时运行的指令

对内存空间的访问范围基本不受限制，不仅能访问用户存储空间，也能访问系统存储空间，特权指令只允许操作系统使用，不允许应用程序使用，否则会引起系统混乱。

2) 非特权指令—在用户态时运行的指令

一般应用程序所使用的都是非特权指令，它只能完成一般性的操作和任务，不能对系统中的硬件和软件直接进行访问，其对内存的访问范围也局限于用户空间。

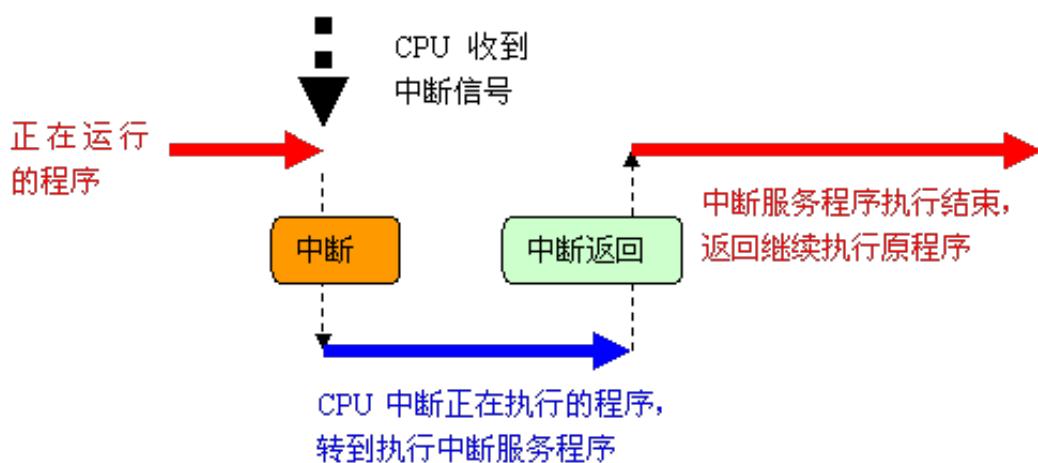
- 用户态切换到内核态的唯一途径——>中断/异常/陷入
- 内核态切换到用户态的途径——>设置程序状态字

注意一条特殊的指令——陷入指令（又称为访管指令，因为内核态也被称为管理态，访管就是访问管理态）该指令给用户提供接口，用于调用操作系统的服务。

3. 什么是中断

还是老问题，为什么在操作系统中设计了中断的概念？为了提高并发执行的效率。

CPU中断正在运行的程序，转到处理中断事件程序



关于中断概念的解释：<http://www.kerneltravel.net/journal/viii/01.htm>

4. 什么是系统调用

定义：系统调用是linux内核中设置了一组用于实现系统功能的子程序。（位于操作系统内核的函数）

系统调用和普通库函数调用非常相似，只是系统调用由操作系统核心提供，运行于核心态，而普通的函数调用由函数库或用户自己提供，运行于用户态。

5. 什么死锁，死锁的条件有哪些?

由于系统中存在一些不可剥夺资源，而当两个或两个以上的进程占有自身资源，并请求对方资源时，会导致每个进程都无法向前推进，这就是死锁。死锁产生的必要条件有四个：互斥条件、不可剥夺条件、请求与保持条件、循环等待条件。

互斥条件指进程要求分配的资源是排他性的，即最多只能同时给一个进程使用。

不剥夺条件是指进程在使用资源完毕之前，资源不能被强行夺走。

请求并保持条件是指进程占有自身本来拥有的资源并要求其他资源。

循环等待条件是指存在一种进程资源的循环等待链。

6. 进程之间有哪些通信方式

1. 信号量(semaphore)

- 信号量 (Semaphore) 由一个值和一个指针组成，指针指向等待该信号量的进程。信号量的值表示相应资源的使用情况。**信号量 $S \geq 0$ 时，S表示可用资源的数量。**
- 从PV操作的角度来解释
 - 执行一次**P操作**意味着请求分配一个资源，因此S的值减1；当S<0时，表示已经没有可用资源，S的绝对值表示当前等待该资源的进程数。请求者必须等待其他进程释放该类资源，才能继续运行。
 - 执行一次**V操作**意味着释放一个资源，因此S的值加1；若S<0，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。注意：**信号量的值只能由PV操作来改变。**
- 从可用进程的角度来解释
 - 信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初始值为1就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。
 - 一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。
 - 当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

2. 管道

- 1) 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
- 2) 匿名管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
- 3) 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。

管道分为pipe（无名管道）和fifo（命名管道）两种，除了建立、打开、删除的方式不同外，这两种管道几乎是一样的。他们都是通过内核缓冲区实现数据传输。

- pipe用于相关进程之间的通信，例如父进程和子进程，它通过pipe()系统调用来创建并打开，当最后一个使用它的进程关闭对他的引用时，pipe将自动撤销。
- FIFO即命名管道，在磁盘上有对应的节点，但没有数据块——换言之，只是拥有一个名字和相应的访问权限，通过mknod()系统调用或者mkfifo()函数来建立的。一旦建立，任何进程都可以通过文件名将其打开和进行读写，而不局限于父子进程，当然前提是进程对FIFO有适当的访问权。当不再被进程使用时，FIFO在内存中释放，但磁盘节点仍然存在。

3. 套接字(Socket)

参考Nginx和PHP之间通过unix socket通信的方式。

4. 信号(signal)

注意信号和信号量的区别

- 信号是Linux系统中用于进程之间通信或操作的一种机制，信号可以在任何时候发送给某一进程，而无须知道该进程的状态。
- 信号来源
 - 硬件来源，例如按下了ctrl+C，通常产生中断信号sigint
 - 软件来源，例如使用系统调用或者命令发出信号。最常用的发送信号的系统函数是kill,raise,setitimer,sigaction,sigqueue函数。

5. 消息队列(message queue)

消息队列，就是一个消息的链表，是一系列保存在内核中消息的列表。用户进程可以向消息队列添加消息，也可以向消息队列读取消息。

6. 共享内存(shared memory)

采用共享内存进行通信的一个主要好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝，对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次：一次从输入文件到共享内存区，另一次从共享内存到输出文件。



图 7.4 共享内存原理

7. 什么是缺页中断？ ★★★★★

在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存时，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

缺页本身是一种中断，与一般的中断一样，需要经过4个处理步骤：

1. 保护CPU现场
2. 分析中断原因
3. 转入缺页中断处理程序进行处理
4. 恢复CPU现场，继续执行

关于缺页中断的实践：<https://liam.page/2017/09/01/page-fault/>

8. 枚举一下常见的（缺页中断）页面置换算法

FIFO算法

LRU算法

置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。

LRU算法计算的精髓在于根据（1）当前物理块中的内容（2）前面计算序列访问了哪些物理块

9. 谈一谈fork()函数

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

- 1) 在父进程中，fork返回新创建子进程的进程ID；
- 2) 在子进程中，fork返回0；
- 3) 如果出现错误，fork返回一个负值；

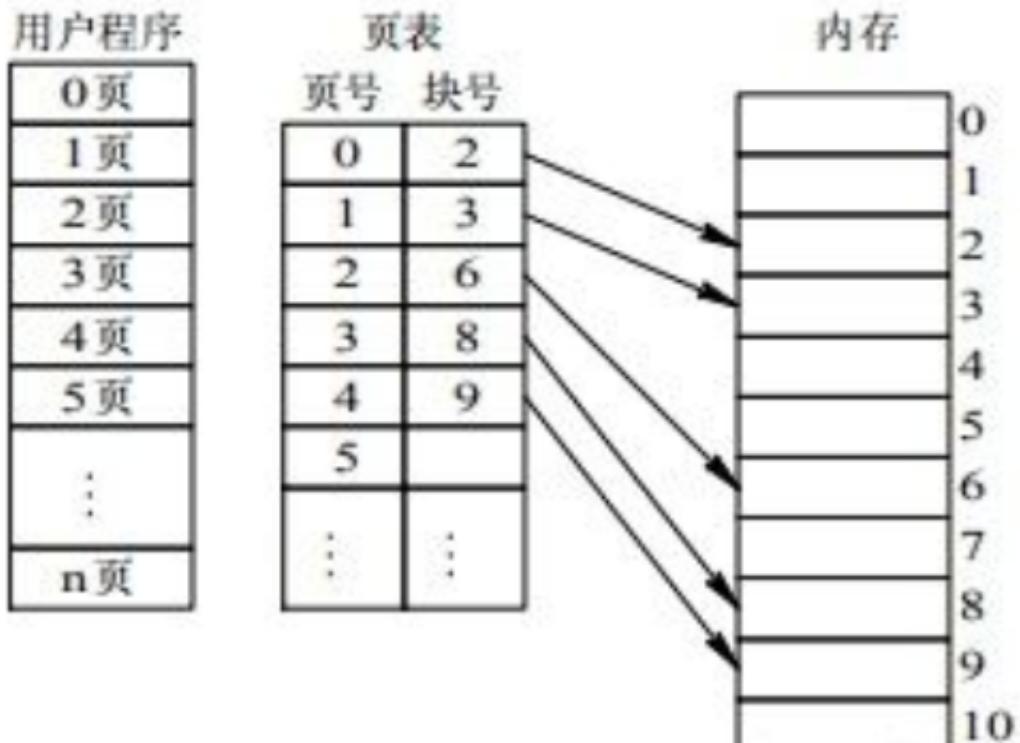
10. 页式内存管理

1. 如何实现逻辑地址到物理地址的映射管理

分页机制的思想是：通过映射，可以使连续的线性地址与物理地址相关联，逻辑上连续的线性地址对应的物理地址可以不连续。分页的作用 - 将线性地址转换为物理地址 - 用大小相同的页替换大小不同的段。

逻辑地址 = 页号 + 页内偏移量

取到页号之后，查询页表，得到块号，然后在内存中通过块号&页内偏移量得到最终的物理地址



11. linux终端按下Ctrl+C发生了什么 ★★★

`ctrl-c`: (`kill foreground process`) 发送 `SIGINT` 信号给前台进程组中的所有进程，强制终止程序的执行；
`ctrl-z`: (`suspend foreground process`) 发送 `SIGTSTP` 信号给前台进程组中的所有进程，常用于挂起一个进程，而非结束进程，用户可以使用 `fg/bg` 操作恢复执行前台或后台的进程。

根据 `setpgrp` manual page 的说法，按下 `ctrl-c` 后（主要是操作系统进程通信-信号机制）：

- 终端产生 `SIGINT` 信号
- 前台进程组中的所有进程都会接收到 `SIGINT` 信号然后退出(默认动作)
- shell通过调用 `waitpid` 清理进程表中子进程信息

12. 内存管理

1. 页式内存管理

1. 固定分区分配

缺点：容易产生内部碎片

2. 动态分区分配

分配算法：

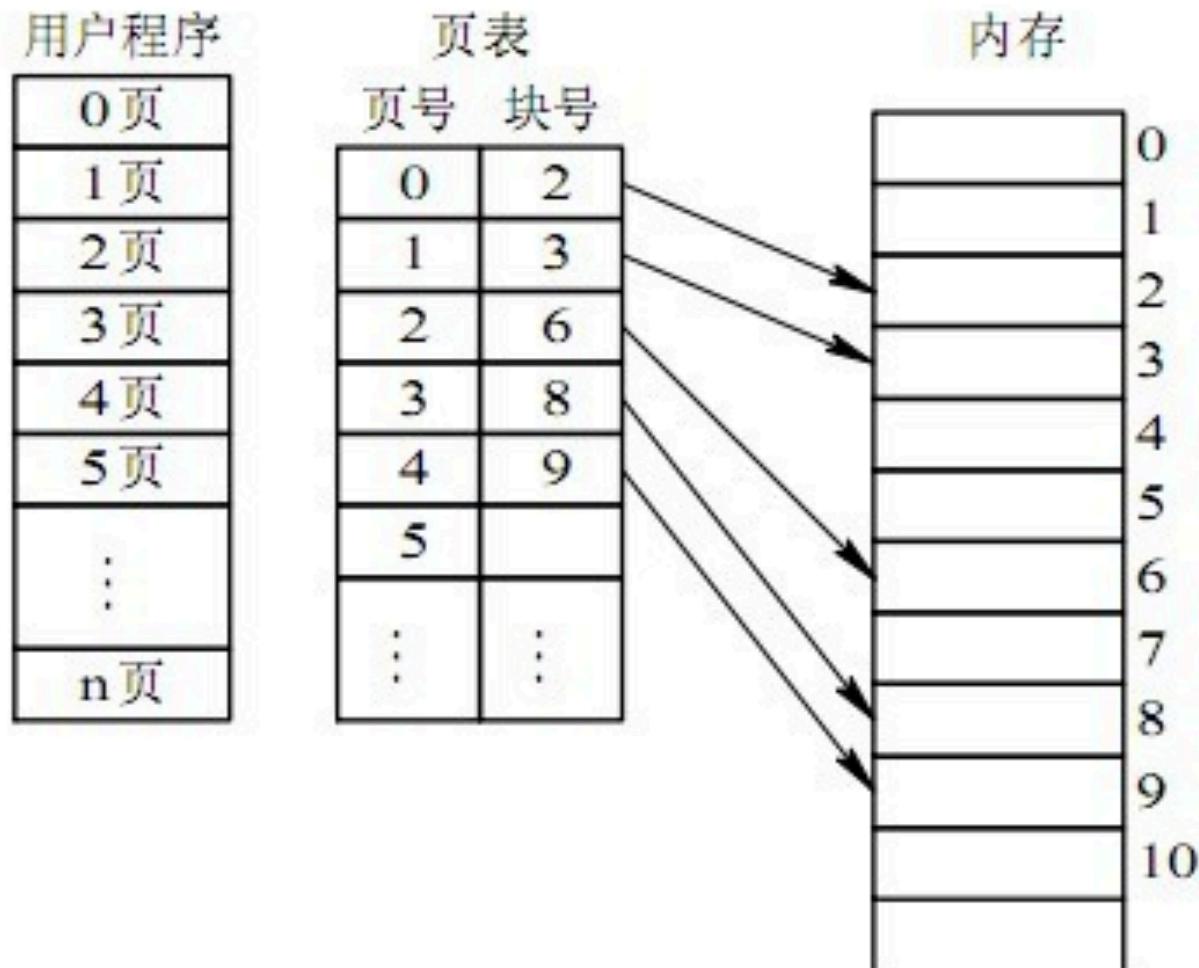
- 最先适配法(First-fit)：按分区在内存的先后次序从头查找，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分会产生较多小分区，每次分配时查找时间开销便会增大。

- 下次适配法(循环首次适应算法 next fit): 按分区在内存的先后次序, 从上次分配的分区起查找(到最后{区时再从头开始}), 找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好, 使空闲分区分布得更均匀, 但较大空闲分区不易保留。
- 最佳适配法(best-fit): 按分区在内存的先后次序从头查找, 找到其大小与要求相差最小的空闲分区进行分配。从个别来看, 外碎片较小; 但从整体来看, 会形成较多外碎片优点是较大的空闲分区可以被保留。
- 最坏适配法(worst-fit): 按分区在内存的先后次序从头查找, 找到最大的空闲分区进行分配。基本不留下小空闲分区, 不易形成外碎片。但由于较大的空闲分区不被保留, 当对内存需求较大的进程需要运行时, 其要求不易被满足。

缺点: 容易产生外部碎片

3. 页式内存管理

将程序的逻辑地址空间划分为固定大小的页(page), 而物理内存划分为同样大小的页框(page frame)。程序加载时, 可将任意一页放入内存中任意一个页框, 这些页框不必连续, 从而实现了离散分配。该方法需要CPU的硬件支持, 来实现逻辑地址和物理地址之间的映射。



CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号, 将物理页框号与页内地址相加形成物理地址(见图4-4)。

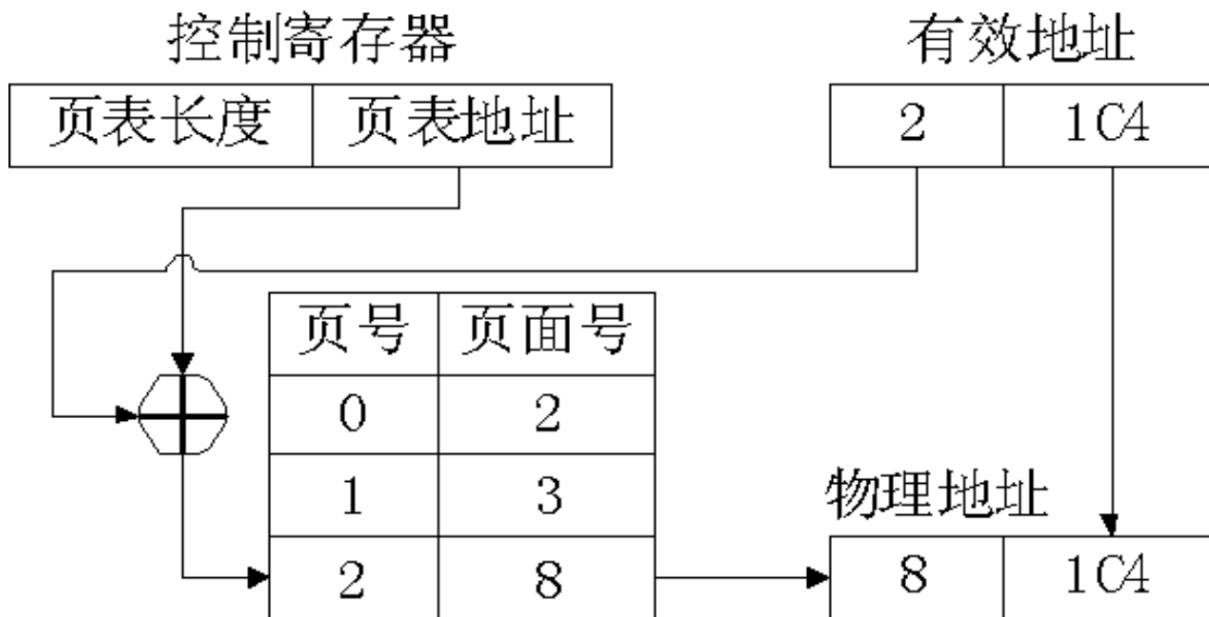


图4-4 页式管理的地址变换

上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存：

- 第一次用来查找页表将操作数的逻辑地址变换为物理地址；
- 第二次完成真正的读写操作。

这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器(例如，快表)中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送入快表，并将此页号与快表中的所有页号进行比较，而且这种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于关联存储器的访问速度比内存的访问速度快得多。

参考：<https://www.hahack.com/wiki/c-memory.html>

2. 段式内存管理

在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址(见图4—5)。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将进程段表的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。

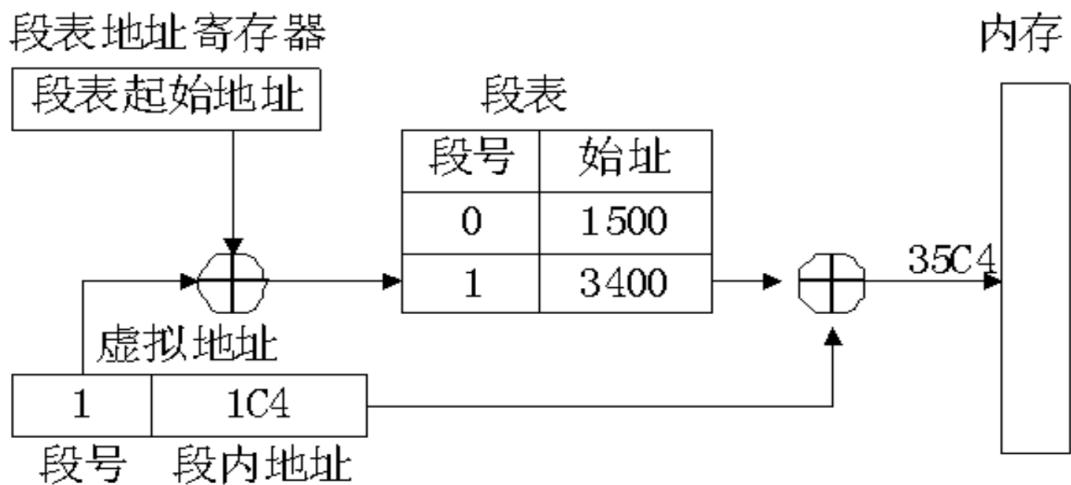


图4—5 段式管理的地址变换

3. 两者之间的区别

页式和段式系统有许多相似之处。比如，两者都采用离散分配方式，且都通过地址映射机构来实现地址变换。但概念上两者也有很多区别，主要表现在：

1)、需求：是信息的物理单位，分页是为了实现离散分配方式，以减少内存的碎片，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地满足用户的需要。

一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。2)、大小：页大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的。段的长度不固定，且决定于用户所编写的程序，通常由编译系统在对源程序进行编译时根据信息的性质来划分。

3)、逻辑地址表示：页式系统地址空间是一维的，即单一的线性地址空间，程序员只需利用一个标识符，即可表示一个地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

4)、比页大，因而段表比页表短，可以缩短查找时间，提高访问速度。

4. 段页式内存管理

分页可以提高内存的利用率，分段从逻辑的角度来看比较方便程序员管理。

13. 编程实现一个LRU算法 ★★★★★

14. 进度调度算法 ★★★★★

注意和页面调度算法区别开来

15. linux下malloc底层实现原理 ★★★★☆

查考实际编程和操作系统底层的结合

<https://azhao.net/index.php/archives/81/>

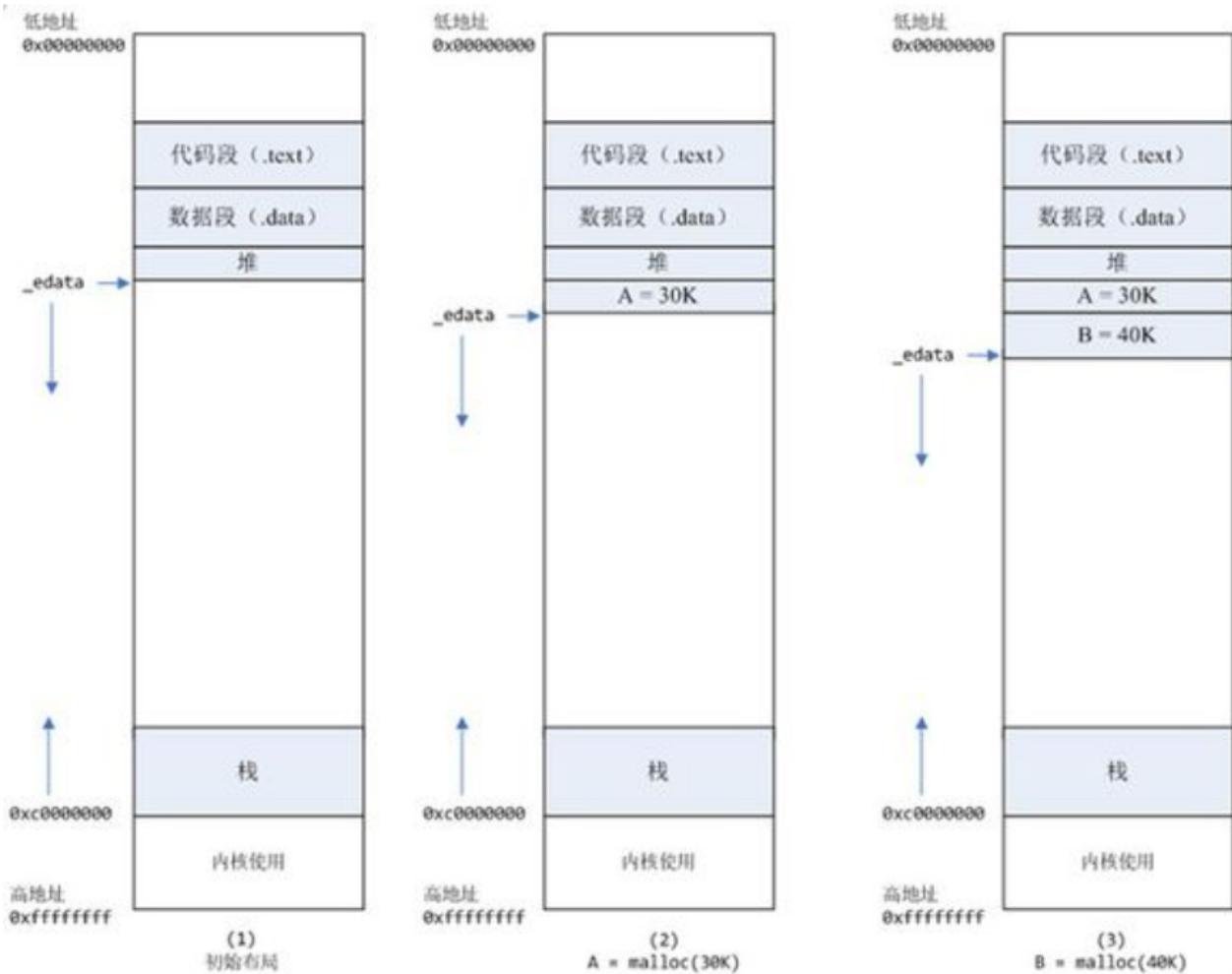
https://m.baidu.com/tc?from=bd_graph_mm_tc&srd=1&dict=20&src=http%3A%2F%2Fwww.th7.cn%2Fsystem%2Flin%2F201609%2F181674.shtml&sec=1563815502&di=a2b471c523e5a3b1

1. C语言内存分布



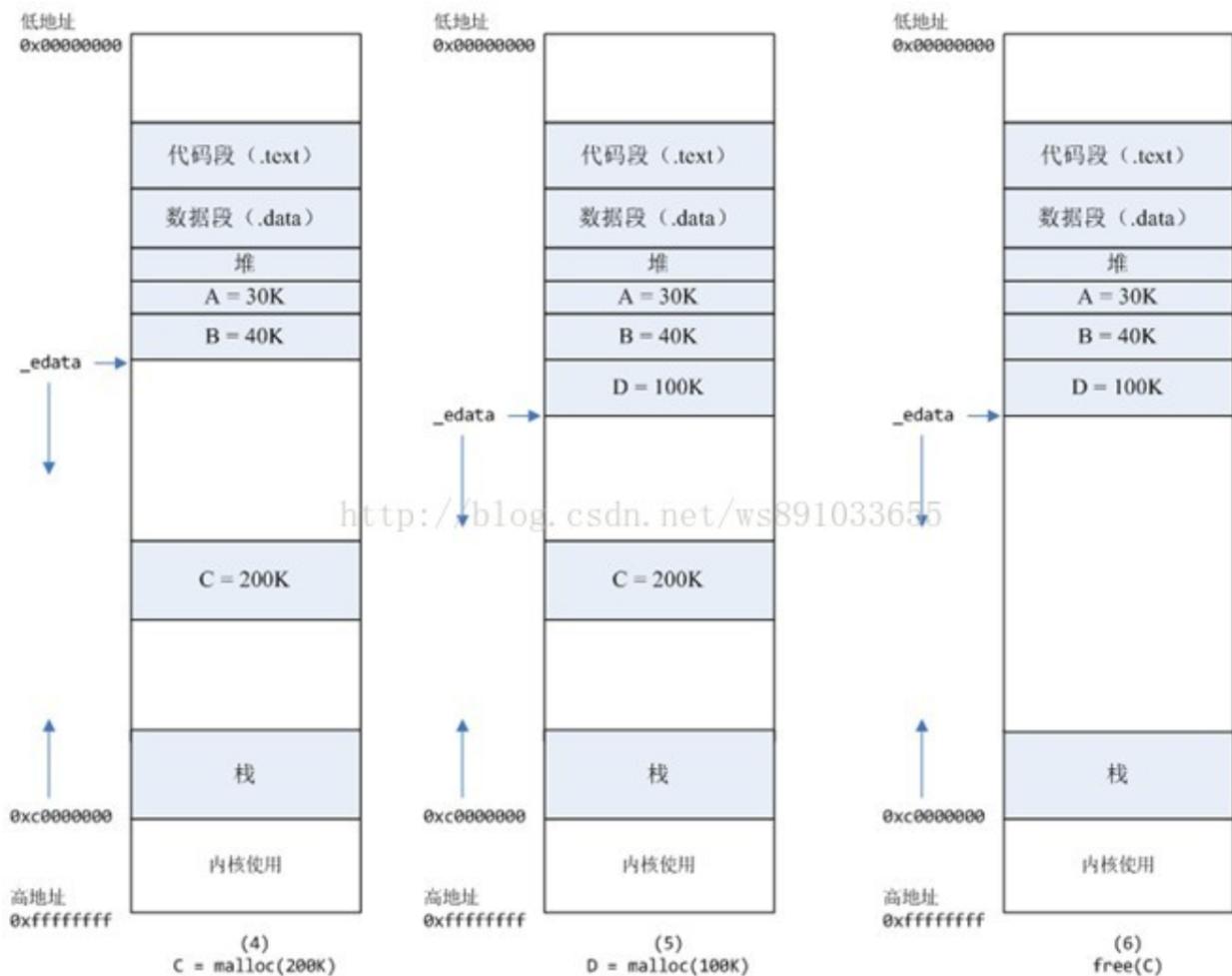
1、栈区（stack） — 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈 2、堆区（heap） — 一般由程序员分配释放（malloc分配内存存在堆），若程序员不释放，程序结束时可能由OS回收。

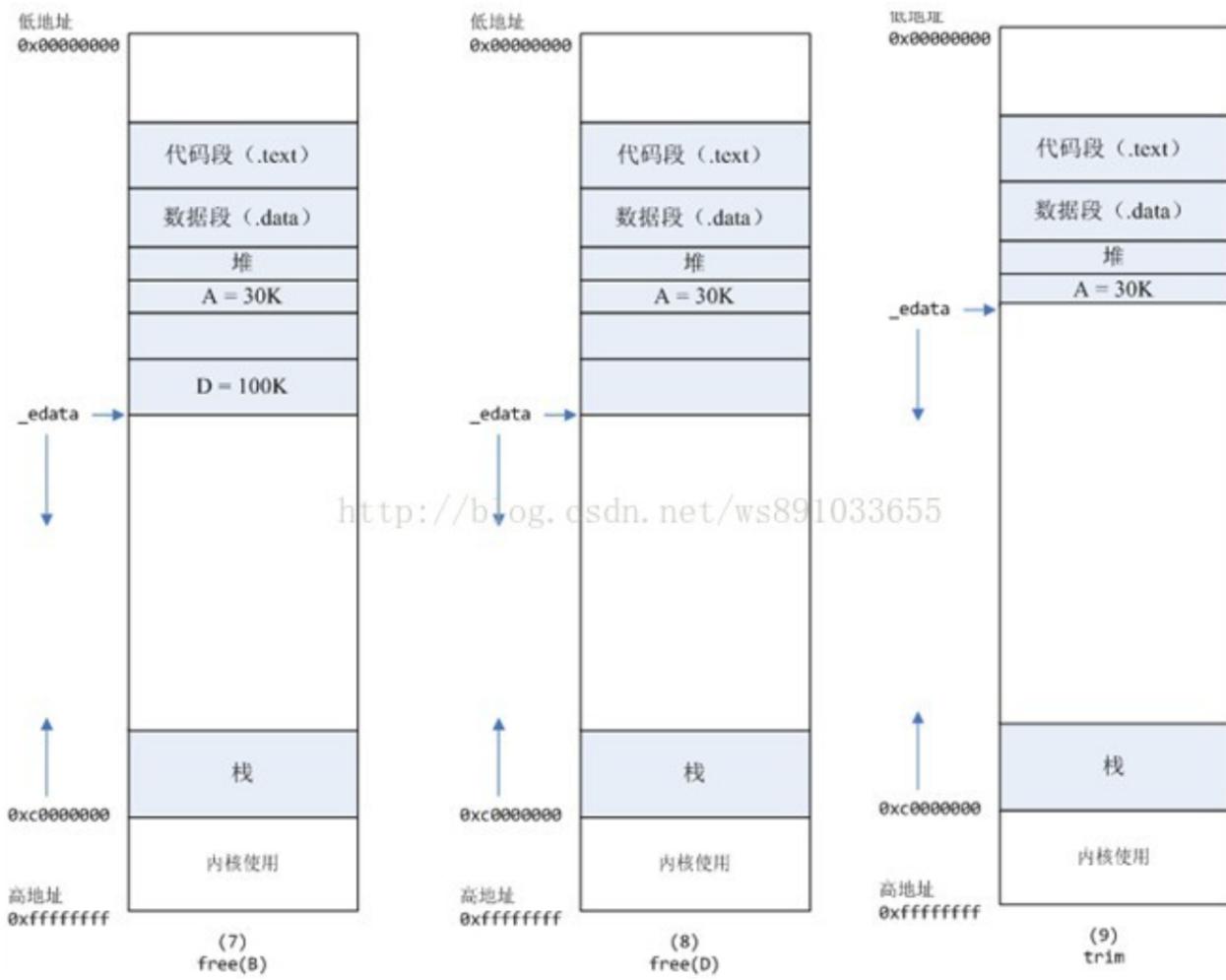
2. 其中堆内存配置的方式



- 情况一、`malloc`小于128k的内存，使用`brk`分配内存，将`_edata`往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)

- 1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。其中，`mmap`内存映射文件是在堆和栈的中间（例如`libc-2.2.93.so`，其它数据文件等），为了简单起见，省略了内存映射文件。
- 2、进程调用`A = malloc(30K)`以后，内存空间如图2：`malloc`函数会调用`brk`系统调用，将`edata`指针往高地址推30K，就完成虚拟内存分配。你可能会问：只要把`edata+30K`就完成内存分配了？事实是这样的，`_edata+30K`只是完成虚拟地址的分配，`A`这块内存现在还是没有物理页与之对应的，等到进程第一次读写`A`这块内存的时候，发生缺页中断，这个时候，内核才分配`A`这块内存对应的物理页。也就是说，如果用`malloc`分配了`A`这块内容，然后从来不访问它，那么，`A`对应的物理页是不会被分配的。
- 3、进程调用`B = malloc(40K)`以后，内存空间如图3





- 4、进程调用C=malloc(200K)以后，内存空间如图4：默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M_MMAP_THRESHOLD选项调节），那就不是去推`_edata`指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。这样子做主要是因为:: brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。
- 5、进程调用D=malloc(100K)以后，内存空间如图5；
- 6、进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。
- 7、进程调用free(B)以后，如图7所示：B对应的虚拟内存和物理内存都没有释放，因为只有一个`_edata`指针，如果往回推，那么D这块内存怎么办呢？当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。
- 8、进程调用free(D)以后，如图8所示：B和D连接起来，变成一块140K的空闲内存。
- 9、默认情况下：当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作(trim)。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩，变成图9所示

16. 操作系统中的常识

1. MMU (Memory Management Unit)

MMU即内存管理单元(Memory Manage Unit)，是一个与软件密切相关的硬件部件

所以物理地址①是通过CPU对外地址总线②传给Memory Chip③使用的地址;而虚拟地址④是CPU内部执行单元⑤产生的,发送给MMU⑥的地址。硬件上MMU⑥一般封装于CPU芯片⑦内部,所以虚拟地址④一般只存在于CPU⑦内部,到了CPU外部地址总线引脚上②的信号就是MMU转换过的物理地址①。

2. 快表 (TLB)

3.

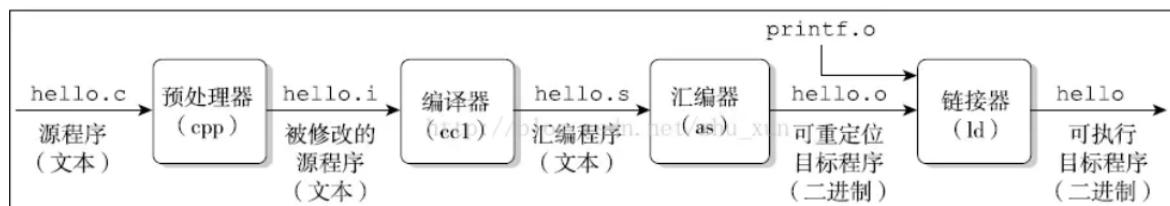
17. IO多路复用 ★★★★☆

1. 高性能IO模型

服务器端编程经常需要构造高性能的IO模型,常见的IO模型有四种:

- (1) 同步阻塞IO (Blocking IO) : 即传统的IO模型。
- (2) 同步非阻塞IO (Non-blocking IO) : 默认创建的socket都是阻塞的,非阻塞IO要求socket被设置为NONBLOCK。注意这里所说的NIO并非Java的NIO (New IO) 库。
- (3) IO多路复用 (IO Multiplexing) : 即经典的Reactor设计模式,有时也称为异步阻塞IO, Java中的Selector和Linux中的epoll都是这种模型。
- (4) 异步IO (Asynchronous IO) : 即经典的Proactor设计模式,也称为异步非阻塞IO。

18. C语言编译的整个过程



一个完整的编译系统与一个用C编写的程序hello.c的编译过程

1. 预处理阶段。主要是处理源文件中以“#”开头的预编译指令。

删除#define并展开宏
处理所有条件预编译指令,如#if, #ifdef, #endif
插入头文件到#include处
删除所有注释
添加行号和文件名标识,以便编译时编译器产生调试用的行号信息
保留所有#pragma编译指令

2. 编译阶段。将预处理得到的预处理文件进行语法分析,词法分析,语义分析,优化后,生成汇编代码文件(汇编语言源程序)。
3. 汇编阶段。利用汇编程序(汇编器)将汇编语言源程序转换成机器指令序列(机器语言程序)。

- 链接阶段。将多个可重定位的目标文件.o合并以生成可执行文件，其可以被加载到内存中，由系统执行。

19. 编译性语言和解释性语言的区别

编译：一次把所有的代码转换成机器语言，然后写成可执行文件

解释：程序每执行到源程序的某一条指令，则会有一个称之为解释程序的外壳程序将源代码转换成二进制代码以供执行（一边解释一遍执行）

20. CPU执行原理-简化

CPU的运行原理就是：

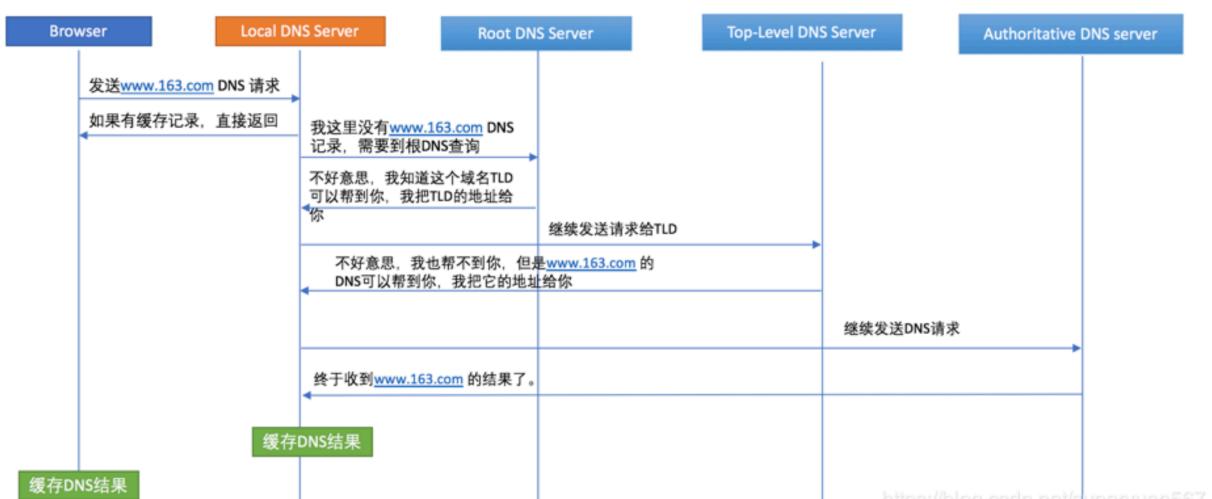
- 取指令：CPU的控制器从内存读取一条指令并放入指令寄存器。指令的格式一般是这个样子滴：操作码就是汇编语言里的mov, add, jmp等符号码；操作数地址说明该指令需要的操作数所在的地方，是在内存里还是在CPU的内部寄存器里。
- 指令译码（解码）：指令寄存器中的指令经过译码，决定该指令应进行何种操作（就是指令里的操作码）、操作数在哪里（操作数的地址）。
- 执行指令（写回），以一定格式将执行阶段的结果简单的写回。运算结果经常被写进CPU内部的暂存器，以供随后指令快速存取。
- 修改指令计数器，决定下一条指令的地址。

0x4 计算机网络

000 综合

1. 从输入 URL 到页面加载完成，中间发生了什么

1. DNS域名解析



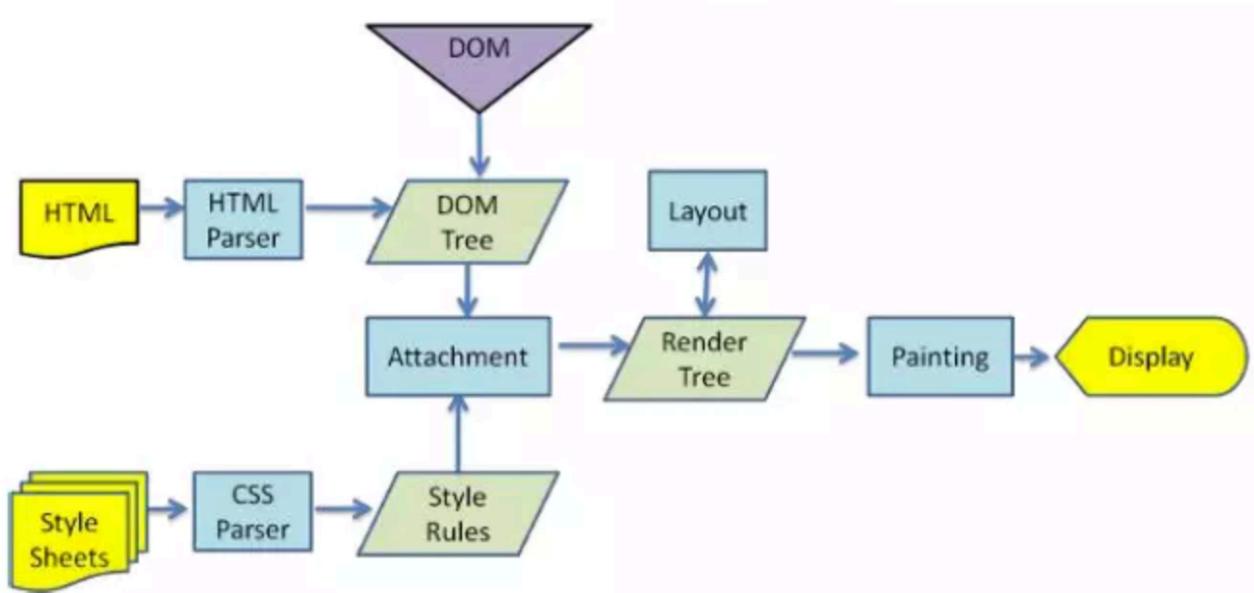
2. 三次握手，建立TCP连接

3. 客户端发送HTTP请求

4. 服务器返回HTTP响应

5. 浏览器渲染页面

浏览器是一个边解析边渲染的过程。首先浏览器解析HTML文件构建DOM树，然后解析CSS文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上。这个过程比较复杂，涉及到两个概念：reflow(回流)和repaint(重绘)。DOM节点中的各个元素都是以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为reflow；当盒模型的位置、大小以及其他属性，如颜色、字体等确定下来之后，浏览器便开始绘制内容，这个过程称为repaint。页面在首次加载时必然会经历 reflow 和 repaint 过程是非常消耗性能的，尤其是在移动设备上，它会破坏用户体验，有时会造成页面卡顿。所以我们应该尽可能少的减少reflow和repaint。



6. 四次挥手，关闭TCP连接

2. 跨域问题

1. 浏览器同源策略

因为浏览器的同源策略规定某域下的客户端在没明确授权的情况下，不能读写另一个域的资源。而在实际开发中，前后端常常是相互分离的，并且前端的项目部署也常常不在一个服务器内或者在一个服务器的不同端口下。前端想要获取后端的数据，就必须发起请求，如果不做一些处理，就会受到浏览器同源策略的约束。后端可以收到请求并返回数据，但是前端无法收到数据。

2. 假设没有同源策略

有一个小小的东西叫cookie大家应该知道，一般用来处理登录等场景，目的是让服务端知道谁发出的这次请求。如果你请求了接口进行登录，服务端验证通过后会在响应头加入Set-Cookie字段，然后下次再发请求的时候，浏览器会自动将cookie附加在HTTP请求的头字段Cookie中，服务端就能知道这个用户已经登录过了。知道这个之后，我们来看场景：

1.你准备去清空你的购物车，于是打开了买买买网站www.maimaimai.com，然后登录成功，一看，购物车东西这么少，不行，还得买多点。

2.你在看有什么东西买的过程中，你的好基友发给你一个链接www.nidongde.com，一脸yin笑地跟你说：“你懂的”，你毫不犹豫打开了。

3.你饶有兴致地浏览着www.nidongde.com，谁知这个网站暗地里做了些不可描述的事情！由于没有同源策略的限制，它向www.maimaimai.com发起了请求！聪明的你一定想到上面的话“服务端验证通过后会在响应头加入Set-Cookie字段，然后下次再发请求的时候，浏览器会自动将cookie附加在HTTP请求的头字段Cookie中”，这样一来，这个不法网站就相当于登录了你的账号，可以为所欲为了！如果这不是一个买买买账号，而是你的银行账号，那.....

3. 解决方案

1. Cross Origin Resource Share (CORS)

CORS是一个跨域资源共享方案，为了解决跨域问题，通过增加一系列请求头和响应头，规范安全地进行跨站数据传输。

请求头主要包括

请求头	解释
Origin	Origin头在跨域请求或预先请求中，标明发起跨域请求的源域名。
Access-Control-Request-Method	Access-Control-Request-Method头用于表明跨域请求使用的实际HTTP方法
Access-Control-Request-Headers	Access-Control-Request-Headers用于在预先请求时，告知服务器要发起的跨域请求中会携带的请求头信息
with-credentials	跨域请求携带cookie

响应头主要包括

响应头	解释
Access-Control-Allow-Origin	Access-Control-Allow-Origin头中携带了服务器端验证后的允许的跨域请求域名，可以是一个具体的域名或是一个*（表示任意域名）。
Access-Control-Expose-Headers	Access-Control-Expose-Headers头用于允许返回给跨域请求的响应头列表，在列表中的响应头的内容，才可以被浏览器访问。
Access-Control-Max-Age	Access-Control-Max-Age用于告知浏览器可以将预先检查请求返回结果缓存的时间，在缓存有效期内，浏览器会使用缓存的预先检查结果判断是否发送跨域请求。
Access-Control-Allow-Methods	Access-Control-Allow-Methods用于告知浏览器可以在实际发送跨域请求时，可以支持的请求方法，可以是一个具体的方法列表或是一个*（表示任意方法）。

SpringBoot解决方案

```

HttpServletResponse httpServletResponse = (HttpServletResponse) response;
String temp = request.getHeader("Origin");
httpServletResponse.setHeader("Access-Control-Allow-Origin", temp);
// 允许的访问方法
httpServletResponse.setHeader("Access-Control-Allow-Methods", "POST,
GET, PUT, OPTIONS, DELETE, PATCH");
//Access-Control-Max-Age 用于 cors 相关配置的缓存
httpServletResponse.setHeader("Access-Control-Max-Age", "3600");
httpServletResponse.setHeader("Access-Control-Allow-Headers",
"Origin, X-Requested-With, Content-Type, Accept,token");
httpServletResponse.setHeader("Access-Control-Allow-Credentials",
"true");

```

3. 计算机网络模型

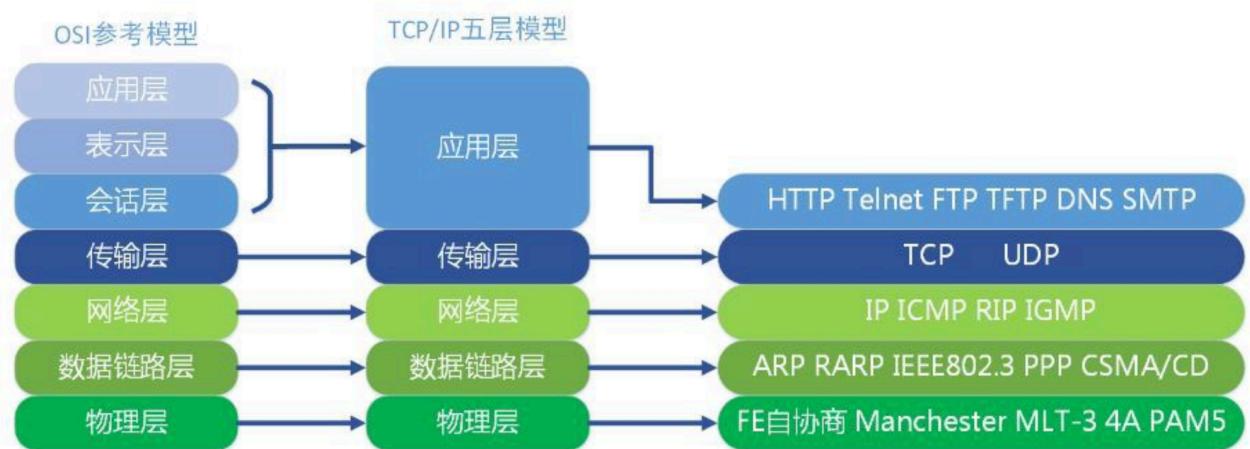
1. OSI模型和TCP/IP五层模型

OSI (Open System Interconnect)，即开放式系统互联。一般都叫OSI参考模型，是ISO（国际标准化组织）组织在1985年研究的网络互连模型。

1. 对应关系和物理设备



2. 对应关系和协议

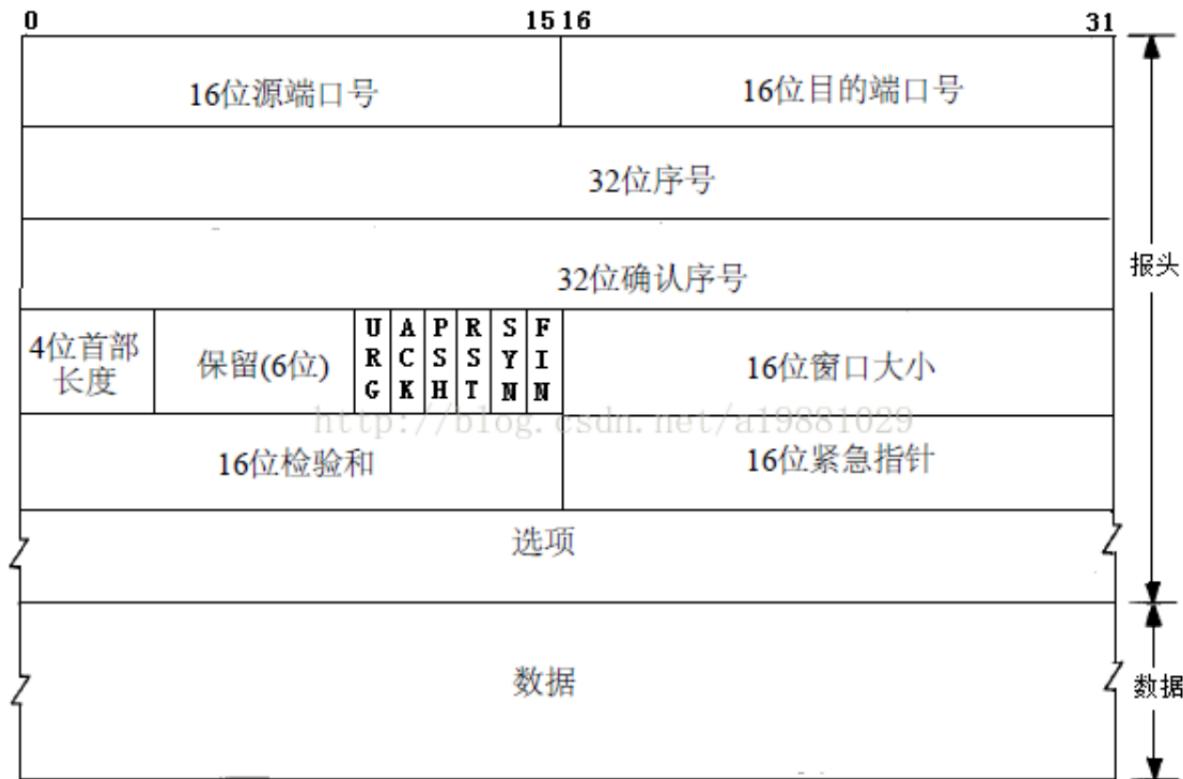


3. 常见协议和端口

001 传输层协议

1. TCP协议从入门到熟悉

1. 报文头部



源端口号和目的端口号：再加上Ip首部的源IP地址和目的IP地址可以唯一确定一个TCP连接

32位序号：表示一次tcp通信过程（从建立连接到断开）过程中某一次传输方向上的字节流的每个字节的编号。假定主机A和B进行tcp通信，A传送给B一个tcp报文段中，序号值被系统初始化为某一个随机值ISN，那么在该传输方向上（从A到B），后续的所有tcp报文段中的序号值都会被设定为ISN加上该报文段所携带数据的第一个字节在整个字节流中的偏移。例如某个TCP报文段传送的数据是字节流中的第1025~2048字节，那么该报文段的序号值就是ISN+1025。

32位确认号：用作对另一方发送的tcp报文段的响应。其值是收到对方的tcp报文段的序号值+1。假定主机A和B进行tcp通信，那么A发出的tcp报文段不但带有自己的序号，也包含了对B发送来的tcp报文段的确认号。反之也一样。

URG-紧急指针有效

ACK-确认序号有效

PSH-接收方应尽快将这个报文交给应用层

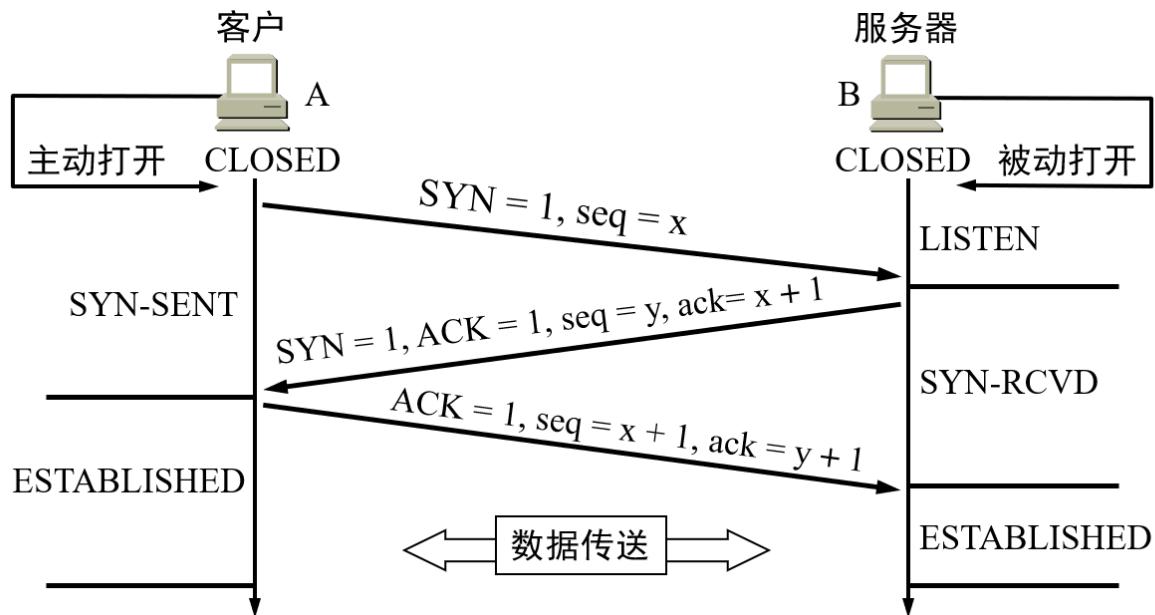
RST-连接重置

SYN-同步序号用来发起一个连接

FIN-终止一个连接

2. 三次握手

关注传输的状态码、序号|确认号以及传输之后的状态【closed listen syn-sent syn-rcvd established】



- 第一次握手(SYN=1, seq=x):

客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号 X,保存在包头的序列号(Sequence Number)字段里。

发送完毕后，客户端进入 **SYN_SEND** 状态。

- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):

服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即 X+1。发送完毕后，服务器端进入 **SYN_RCVD** 状态。

- 第三次握手(ACK=1, ACKnum=y+1)

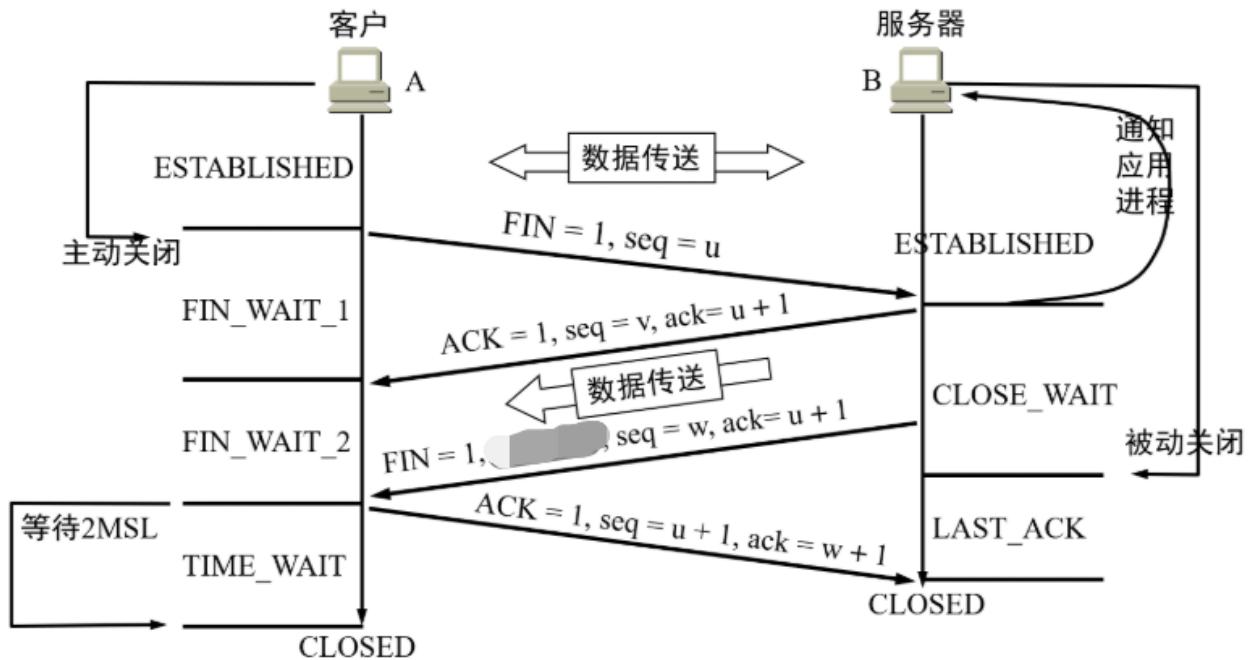
客户端再次发送确认包(ACK)， SYN 标志位为0， ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写seq+1

发送完毕后，客户端进入 **ESTABLISHED** 状态，当服务器端接收到这个包时，也进入 **ESTABLISHED** 状态，TCP 握手结束。

3. 四次挥手

client : established fin_wait_1 fin_wait_2 time_wait

server: established close_wait, last_ack



- 1) 客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部， $\text{FIN}=1$ ，其序列号为 $\text{seq}=u$ （等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入FIN-WAIT-1（终止等待1）状态。TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。
- 2) 服务器收到连接释放报文，发出确认报文， $\text{ACK}=1$, $\text{ack}=u+1$ ，并且带上自己的序列号 $\text{seq}=v$ ，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。
- 3) 客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。
- 4) 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文， $\text{FIN}=1$, $\text{ack}=u+1$ ，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 $\text{seq}=w$ ，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。
- 5) 客户端收到服务器的连接释放报文后，必须发出确认， $\text{ACK}=1$, $\text{ack}=w+1$ ，而自己的序列号是 $\text{seq}=u+1$ ，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过 $2 \times MSL$ （最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。
- 6) 服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

4. 细节分析【建立连接和关闭连接】

1. 为什么TIME_WAIT状态需要经过2MSL才能返回到CLOSE状态？

所谓的2MSL是两倍的MSL(Maximum Segment Lifetime/最大报文段生存时间)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

1. 为什么不会直接进入closed状态，而是进入time_wait状态

我们知道，TCP是比较可靠的。当TCP向另一端发送数据时，他要求对端返回一个确认（如同我们关闭时候的FIN和ACK）。如果没有收到确认，则会重发。

回忆一下我们最终的那个FIN与ACK，被动关闭方发送FIN，并等待主动关闭方返回的ACK。我们假设最终的ACK丢失，被动关闭方将需要重新发送它的最终那个FIN，主动关闭方必须维护状态信息（TIME_WAIT），以允许它重发最终的那个ACK。

如果没有了这个状态，当他第二次收到FIN时，会响应一个RST（也是一种类型的TCP分节），会被服务器解释成一个错误。

为了TCP打算执行必要的工作以彻底终止某个连接两个方向上的数据流（即全双工关闭），那么他必须要正确处理连接终止四个分节中任何一个分节丢失的情况。

2. 为什么time_wait的时间是2MSL，不是MSL

首先，存在这样的情况，某个路由器崩溃或者两个路由器之间的某个链接断开时，路由协议需要花费数秒到数分钟的时间才能稳定找出另一条通路。在这段时间内，可能发生路由循环（路由器A把分组发送给B，B又发送回给A），这种情况我们称之为迷途。假设迷途的分组是一个TCP分节，在迷途期间，发送端TCP超时并重传该分组，重传分组通过某路径到达目的地，而后不久（最多MSL秒）路由循环修复，早先迷失在这个循环中的分组最终也被送到目的地。这种分组被称之为重复分组或者漫游的重复分组，TCP必须要正确处理这些重复的分组。

我们假设ip1:port1和ip2:port2之间有一个TCP连接。我们关闭了这个链接，过一段时间后在相同IP和端口之间建立了另一个连接。**TCP必须防止来自之前那个连接的老的重复分组在新连接上出现。**为了做到这一点，**TCP将不复用处于TIME_WAIT状态的连接。**2MSL的时间足以让某个方向上的分组存活MSL秒后被丢弃，另一个方向上的应答也最多存活MSL秒后被丢弃。

Client会在发出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。

2. 为什么是三次握手，假如是两次握手呢？

谢希仁版《计算机网络》中的例子是这样的，“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

3. 为什么需要四次挥手，假如是三次挥手呢？【close_wait时间段的作用】

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，“你发的FIN报文我收到了”。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

4. 什么是SYN攻击

在三次握手过程中，服务器发送SYN-ACK之后，收到客户端的ACK之前的TCP连接称为半连接(half-open connect)。此时服务器处于SYN_RCVD状态。当收到ACK后，服务器才能转入ESTABLISHED状态。

SYN 攻击指的是，攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送SYN包，服务器回复确认包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。

5. RST位有什么作用

RST表示复位，用来异常的关闭连接，在TCP的设计中它是不可或缺的。就像上面说的一样，发送RST包关闭连接时，不必等缓冲区的包都发出去（不像上面的FIN包），直接就丢弃缓存区的包发送RST包。而接收端收到RST包后，也不必发送ACK包来确认。

TCP处理程序会在自己认为的异常时刻发送RST包。例如，A向B发起连接，但B之上并未监听相应的端口，这时B操作系统上的TCP处理程序会发RST包。

又比如，AB正常建立连接了，正在通讯时，A向B发送了FIN包要求关连接，B发送ACK后，网断了，A通过若干原因放弃了这个连接（例如进程重启）。网通了后，B又开始发数据包，A收到后表示压力很大，不知道这野连接哪来的，就发了个RST包强制把连接关了，B收到后会出现connect reset by peer错误。

5. 如何实现可靠传输

1. 滑动窗口

1. 滑动窗口作用

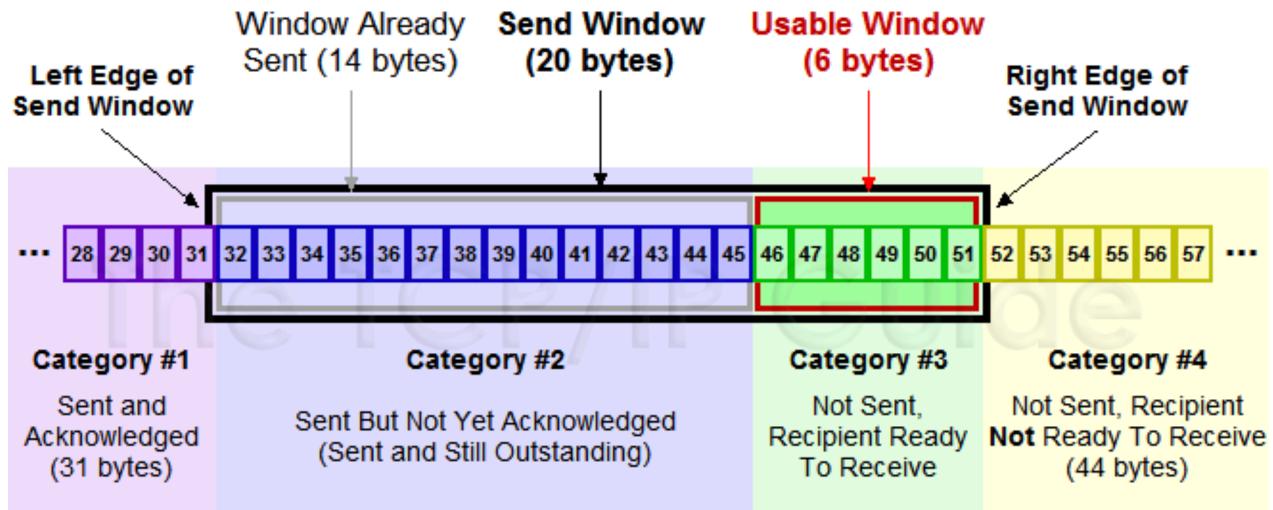
1. Reliability，提供TCP的可靠性，TCP的传输要保证数据能够准确到达目的地，如果不能，需要能检测出来并且重新发送数据。
2. Data Flow Control，提供TCP的流控特性，管理发送数据的速率，不要超过设备的承载能力

2. 滑动窗口组成



1. Sent and Acknowledged：这些数据表示已经发送成功并已经被确认的数据，比如图中的前31个bytes，这些数据其实的位置是在窗口之外了，因为窗口内顺序最低的被确认之后，要移除窗口，实际上是窗口进行合拢，同时打开接收新的带发送的数据
2. Send But Not Yet Acknowledged：这部分数据称为发送但没有被确认，数据被发送出去，没有收到接收端的ACK，认为并没有完成发送，这个属于窗口内的数据。
3. Not Sent, Recipient Ready to Receive：这部分是尽快发送的数据，这部分数据已经被加载到缓存中，也就是窗口中了，等待发送，其实这个窗口是完全有接收方告知的，接收方告知还是能够接受这些包，所以发送方需要尽快的发送这些包
4. Not Sent, Recipient Not Ready to Receive：这些数据属于未发送，同时接收端也不允许发送的，因为这些数据已经超出了发送端所接收的范围

3. 滑动窗口案例



1. Send Window : 20个bytes 这部分值是有接收方在三次握手的时候进行通告的，同时在接收过程中也不断的通告可以发送的窗口大小，来进行适应
2. Window Already Sent: 已经发送的数据，但是并没有收到ACK。

TCP并不是每一个报文段都会回复ACK的，可能会对两个报文段发送一个ACK，也可能会对多个报文段发送1个ACK【累计ACK】，比如说发送方有1/2/3 3个报文段，先发送了2,3 两个报文段，但是接收方期望收到1报文段，这个时候2,3报文段就只能放在缓存中等待报文1的空洞被填上，如果报文1，一直不来，报文2/3也将被丢弃，如果报文1来了，那么会发送一个ACK对这3个报文进行一次确认。

举一个例子来说明一下滑动窗口的原理：

1. 假设32~45 这些数据，是上层Application发送给TCP的，TCP将其分成四个Segment来发往interne
2. seg1 32~34 seg3 35~36 seg3 37~41 seg4 42~45 这四个片段，依次发送出去，此时假设接收端之接收到了seg1 seg2 seg4
3. 此时接收端的行为是回复一个ACK包说明已经接收到了32~36的数据，并将seg4进行缓存（保证顺序，产生一个保存seg3 的hole）
4. 发送端收到ACK之后，就会将32~36的数据包从发送并没确认切到发送已确认，提出窗口，这个时候窗口向右移动
5. 假设接收端通告的Window Size仍然不变，此时窗口右移，产生一些新的空位，这些是接收端允许发送的范畴
6. 对于丢失的seg3，如果超过一定时间，TCP就会重新传送（重传机制），重传成功会seg3 seg4一块被确认，不成功，seg4也将被丢弃

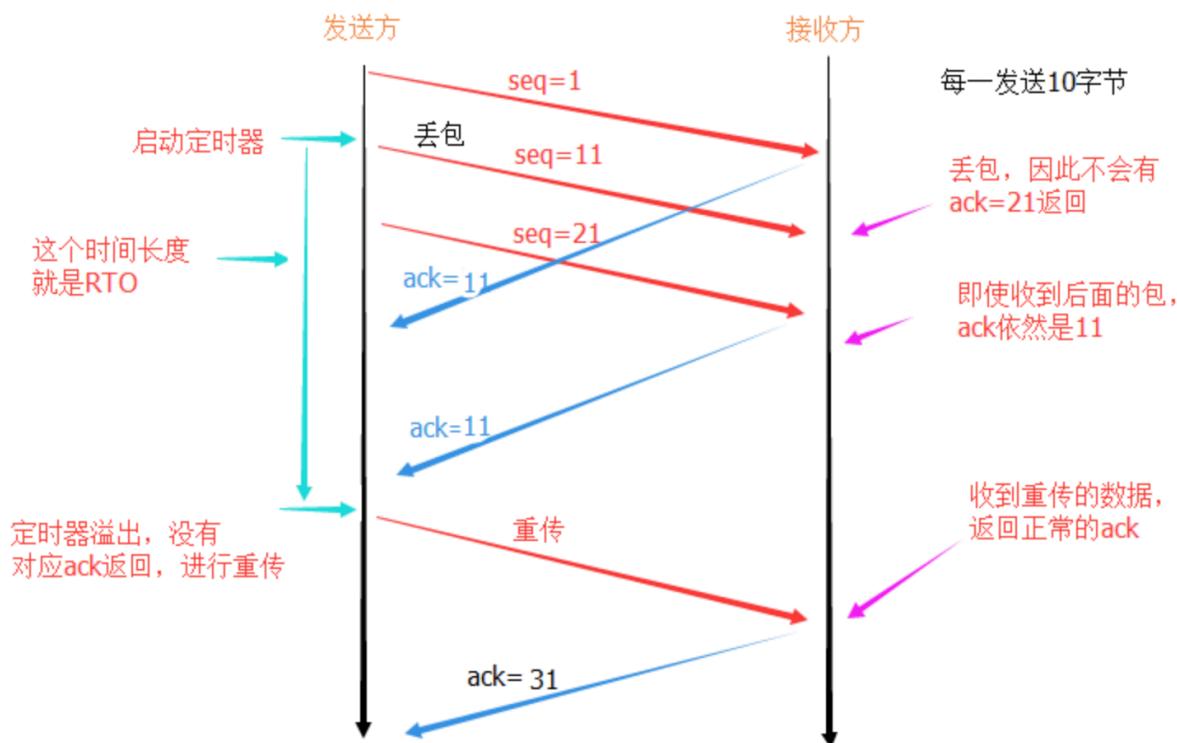
就是不断重复着上述的过程，随着窗口不断滑动，将整个数据流发送到接收端，实际上接收端的Window Size通告也是会变化的，接收端根据这个值来确定何时及发送多少数据，从而对数据流进行流控。

2. 超时重传

就是发送端死等接收端的ack，直到发送端超时之后，在发送一个包，直到收到接收端的ack为止。

例如：接收端给发送端的Ack确认只会确认最后一个连续的包，比如，发送端发了1,2,3,4,5一共五份数据，接收端收到了1, 2，于是回ack 3，然后收到了4（注意此时3没收到），此时的TCP会怎么办？等待发送端的ACK 3，直到超时后，就会再发送3。

当发送端发送数据，发生丢包时，则丢掉的包的ACK一直不会返回。此时发送端就一直等那个ACK返回，若超时，则重传该数据包。对于超时时间RTO（Retransmission TimeOut），有很多复杂的算法。RTO的选择很重要，选短了，可能只是返回时间长但并未丢包，却当做丢包。选长了，迟迟不发丢的包也是个问题。



1. 画出通信图
2. 注意每次传输10个字节【因为滑动窗口存在，不会只是一个字节】
3. 主要是注意seq和ack的联系
4. 定时器timer

3. 拥塞控制

1. 为什么需要拥塞控制

网络中的路由器会有一个数据包处理队列，当路由器接收到的数据包太多而一下子处理不过来时，就会导致数据包处理队列过长。此时，路由器就会无条件的丢弃新接收到的数据封包。

这就会导致上层的TCP协议以为数据包在网络中丢失，进而重传这些数据包，而路由器又会丢弃这些重传的数据包，如此以往，就会导致网络性能急剧下降，引起网络瘫痪。

因此，TCP需要控制数据包发送的数量来避免网络性能的下降。

1. 宽带速率1Gb/s，网络只有两台机器，从一台主机传送数据到另一台，这需要流量控制，以保证接收方能正常接收数据。
2. 宽带速率1Gb/s，网络中有成千上万台机器，几万台主机发送到另外几万台，这需要拥塞控制，不然网络会瘫痪。

2. 拥塞控制算法

001 慢开始

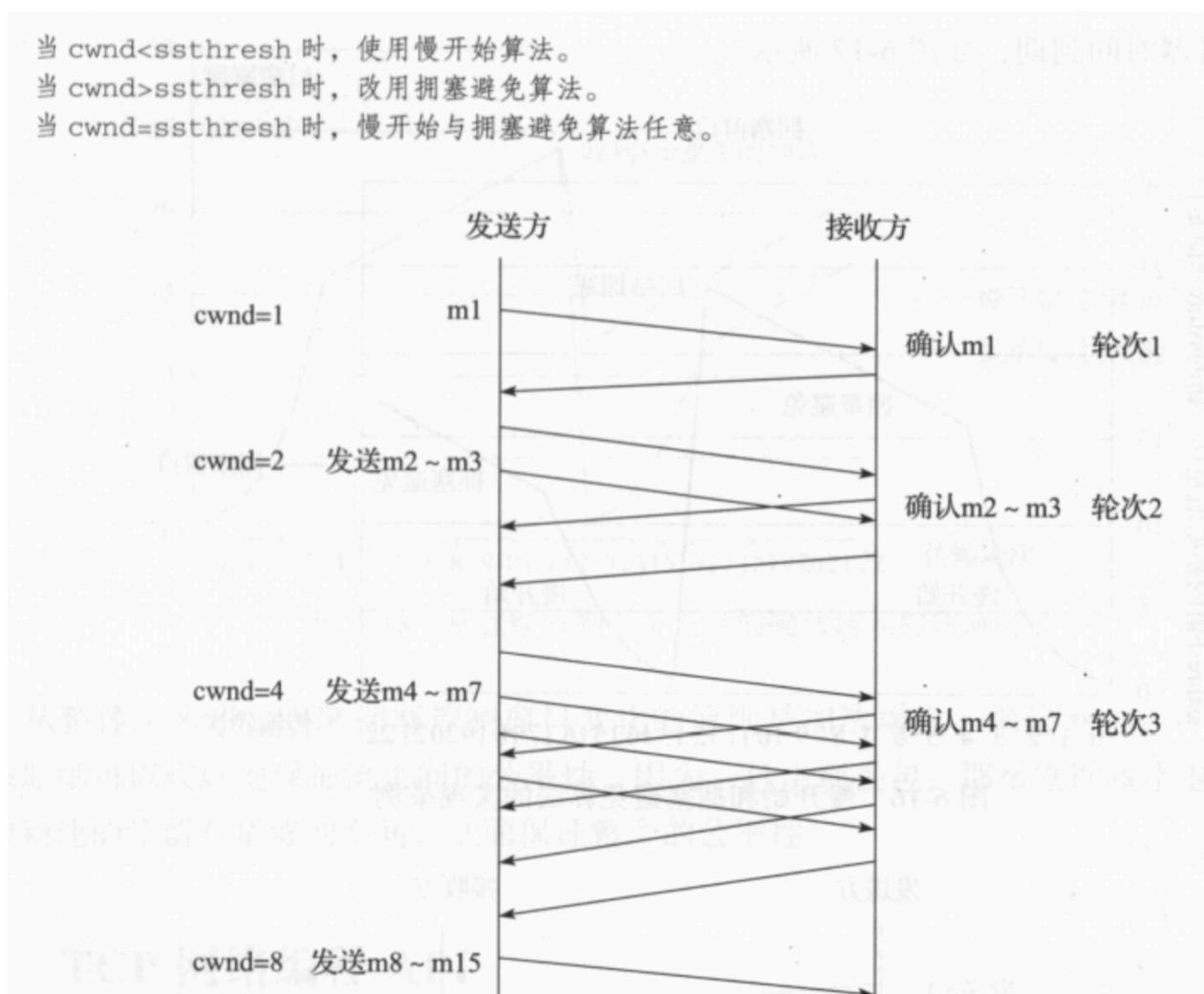
$cwnd=1$ (为方便理解, 这里用报文段的个数作为窗口大小的单位), 在收到接收方发来的确认后 (也就是下个传输轮次), 设置 $cwnd=2$, 然后将发送窗口的数据发送出去。在一次收到接收方发来的确认后, 发送方设置 $cwnd=4$, 再讲发送窗口中的数据发送出去。然后再重复上面的过程。

这里就应该清楚, 慢开始算法中的慢不是说 **cwnd** 增长的慢, 而是相对一下子发送大量数据而言, 这种一次先发送少量的数据包的方式要慢许多。

当然, $cwnd$ 的大小肯定不可能一直以这种指数的方式增长下去, 要不然很快就会增长到引起网络瘫痪的程度了。

所以, 经过一定时间或条件, 我们就要换成拥塞避免算法来发送数据。

当 $cwnd < ssthresh$ 时, 使用慢开始算法。
当 $cwnd > ssthresh$ 时, 改用拥塞避免算法。
当 $cwnd = ssthresh$ 时, 慢开始与拥塞避免算法任意。

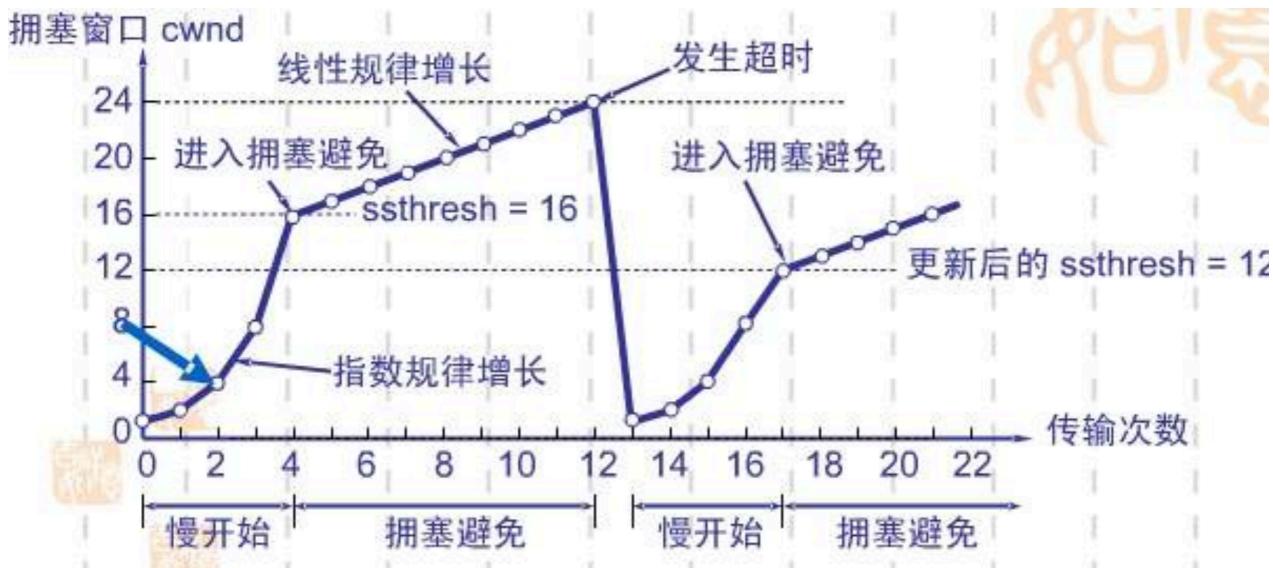


002 拥塞避免

让拥塞窗口 $cwnd$ 缓慢的增大, 即每经过一个往返时间RTT就把发送方的拥塞窗口 $cwnd$ 加1, 而不是加倍, 这样拥塞窗口 $cwnd$ 按线性规律缓慢的增长, 比慢开始算法的拥塞窗口增长速率缓慢的多。

003 发生拥塞

无论是慢启动算法还是拥塞避免算法, 只要判断网络出现拥塞, 就要把慢启动开始门限 (**ssthresh**) 设置为发送窗口的一半 ($>=2$), **cwnd**设置为1, 然后再使用慢启动算法, 这样做的目的能迅速的减少网络当中的数据传输, 使发生拥塞的路由器能够把队列中堆积的分组处理完毕。拥塞窗口是按照线性的规律增长。



004 快重传

这个机制不以时间为驱动，而是以数据来重传！如果接收端包收包没有连续到达，就ACK最后那个可能被丢了的包，如果发送方连续收到接收端3次相同的ack，就重传。

例如：如果发送方发出了1, 2, 3, 4, 5份数据，第一份先到送了，于是就ack回2，结果2因为某些原因没收到，3到达了，于是还是ack回2，后面的4和5都到了，但是还是ack回2，因为2还是没有收到，于是发送端收到了三个ack=2的确认，知道了2还没有到，于是就马上重转2。然后，接收端收到了2，此时因为3, 4, 5都收到了，于是ack回6。

005 快恢复

由于发送方现在认为网络很可能没有发生阻塞，因此现在不执行慢启动算法，而是把cwnd值设置为慢启动门限减半后的值，然后开始执行拥塞避免算法，拥塞窗口cwnd值线性增大。

4. 数据校验【校验和机制】

5. 一些问题

1. 流量控制和拥塞控制有什么区别

- 流量控制：如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。
 - TCP的流量控制是利用滑动窗口机制实现的，接收方在返回的数据中会包含自己的接收窗口的大小，以控制发送方的数据发送。
- 拥塞控制：拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。
- 两者的区别：流量控制是为了预防拥塞。如：在路上行车，交警跟红绿灯是流量控制，当发生拥塞时，如何进行疏散，是拥塞控制。流量控制是指对点通信量的控制。而拥塞控制是全局性的，涉及到所有的主机和降低网络性能的因素。【流量控制是点与点之间，拥塞控制是全局的网络】

2. 拥塞窗口和滑动窗口有什么区别

3. 超时重传和快速重传有什么区别

- 超时重传：超过一定的时间【RTO】才会发生重传

- 快速重传：连续接受到三次相同ACK会发生重传

计算机网络细节补充：<http://www.52im.net/thread-515-1-1.html>

2. TCP和UDP协议的区别。

- 1、TCP面向连接（如打电话要先拨号建立连接）；UDP是无连接的，即发送数据之前不需要建立连接
- 2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP尽最大努力交付，即不保证可靠交付
- 3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流；UDP是面向报文的
 UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）
- 4、每一条TCP连接只能是点到点的；UDP支持一对一，一对多，多对一和多对多的交互通信
- 5、TCP首部开销20字节；UDP的首部开销小，只有8个字节
- 6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

1. TCP和UDP的应用场景

UDP协议比TCP协议的效率更高

- 什么时候使用TCP
 - 当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如**HTTP**、**HTTPS**、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。
- 什么时候使用UDP
 - 对通讯质量要求不严的场景：QQ视频、语音、DNS协议

2. 怎么理解TCP的面向连接和UDP的无连接（不面向连接）

- 实际上就是在客户端和服务端都维护一个变量，这个变量维护现在数据传输的状态，例如传输了哪些数据，下一次需要传输哪些数据，等等，并不是真的我们想象中的真的有什么东西连接着这两端，因为无论对于有连接还是无连接，都有网线连着呢(不包括无线网)，所以连接根本就不是是否真的有什么东西把他们连接起来，真实的含义就是我上面说的，两边维护一个状态变量。
- **UDP通讯有四个参数：源IP、源端口、目的IP和目的端口。而TCP通讯至少有六个参数：源IP、源端口、目的IP和目的端口，以及序列号和应答号。** 序列号和应答号是TCP通讯特有的参数，TCP通讯利用序列号和应答号来保持和确认数据的关联与正确性，是在三次握手中确定的，不正确的序列号和应答号会导致无法正常通讯。因此对TCP连接的连接概念可以简单理解成为同UDP通讯相比，用序列号和应答号确定了相互之间的连接特征，来保证数据传输的正确性。

002 应用层协议

1. HTTP是有状态还是无状态？ TCP是有状态还是无状态？

http为什么要设计成无状态的

tcp为什么是有状态的？可以结合三次握手和四次挥手来说明

HTTP无状态协议，是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

TCP协议是一种有状态协议，因为它是什么，而不是因为它是通过IP使用的，或者因为HTTP构建在它之上。TCP以窗口大小的形式维护状态(端点告知彼此准备好接收多少数据)和数据包顺序(端点必须在从另一个接收数据包时彼此确认).这个状态(另一个人可以接收多少字节,以及他是否接收到最后一个数据包)允许TCP甚至在固有的非可靠协议上是可靠的.因此,TCP是一种有状态协议,因为它需要状态才有用

2. HTTP常用状态码

100 (Continue/继续)

如果服务器收到头信息中带有100-continue的请求，这是指客户端询问是否可以在后续的请求中发送附件。在这种情况下，服务器用100(SC_CONTINUE)允许客户端继续或用417(Expectation Failed)告诉客户端不同意接受附件。这个状态码是HTTP 1.1中新加入的。

200 (OK/正常)

200 (SC_OK)的意思是一切正常。一般用于相应GET和POST请求。这个状态码对servlet是缺省的；如果没有调用setStatus方法的话，就会得到200。

201 (Created/已创建)

201 (SC_CREATED)表示服务器在请求的响应中建立了新文档；应在定位头信息中给出它的URL。

301 (Moved Permanently/永久移动)

被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。

注意：对于某些使用HTTP/1.0协议的浏览器，当它们发送的POST请求得到了一个301响应的话，接下来的重定向请求将会变成GET方式。

301使用场景

1.域名到期不想续费（或者发现了更适合网站的域名），想换个域名。

2.在搜索引擎的搜索结果中出现了不带www的域名，而带www的域名却没有收录，这个时候可以用301重定向来告诉搜索引擎我们目标的域名是哪一个。

3.空间服务器不稳定，换空间的时候。

注：另外，返回301请求码进行跳转被谷歌认为是将网站地址由HTTP迁移到HTTPS的最佳方法(然而大家都用302。。。)

302 Found (Moved Temporarily/临时移动)

要求客户端执行临时重定向（原始描述短语为“Moved Temporarily”）。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。

401 (Unauthorized/未授权)

您的Web服务器认为，客户端发送的HTTP数据流是正确的，但进入网址(URL)资源，需要用户身份验证，而相关信息1)尚未被提供，或2)已提供但没有通过授权测试。这就是通常所知的“HTTP基本验证”。需客户端提供的验证请求在HTTP协议中被定义为WWW-验证标头字段(WWW-Authenticate header field)

桌面应用程序一般不会使用cookie，而是把“用户名+冒号+密码”用BASE64编码的字符串放在http request中的header Authorization中发送给服务端，这种方式叫HTTP基本认证(Basic Authentication)。

403 (Forbidden/拒绝访问)

该状态表示服务器理解了本次请求但是拒绝执行该任务，该请求不该重发给服务器。在HTTP请求的方法不是“HEAD”，并且服务器想让客户端知道为什么没有权限的情况下，服务器应该在返回的信息中描述拒绝的理由。在服务器不想提供任何反馈信息的情况下，服务器可以用404 Not Found代替403 Forbidden。

404 (Not Found/未找到)

500 (Internal Error/内部错误)

3. POST和GET方法区别

直接分析HTTP报文，以及报文常见内容解析





1. POST 方法比 GET 方法安全?

按照网上大部分文章的解释，POST 比 GET 安全，因为数据在地址栏上不可见。

然而，从传输的角度来说，他们都是不安全的，因为 HTTP 在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文。

要想安全传输，就只有加密，也就是 HTTPS。

2. GET 方法的长度限制是怎么回事？

在网上看到很多关于两者区别的文章都有这一条，提到浏览器地址栏输入的参数是有限的。

首先说明一点，HTTP 协议没有 Body 和 URL 的长度限制，对 URL 限制的大多是浏览器和服务器的原因。

浏览器原因就不说了，服务器是因为处理长 URL 要消耗比较多的资源，为了性能和安全（防止恶意构造长 URL 来攻击）考虑，会给 URL 长度加限制。

4. HTTP和HTTPS的区别。★★★★★

HTTPS和HTTP的区别主要如下：

- 1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- 4、http的连接很简单，是无状态的；HTTPS协议是由SSL (Secure Sockets Layer) +HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

5. HTTPS加密原理 ★★★

1. 非对称加密算法

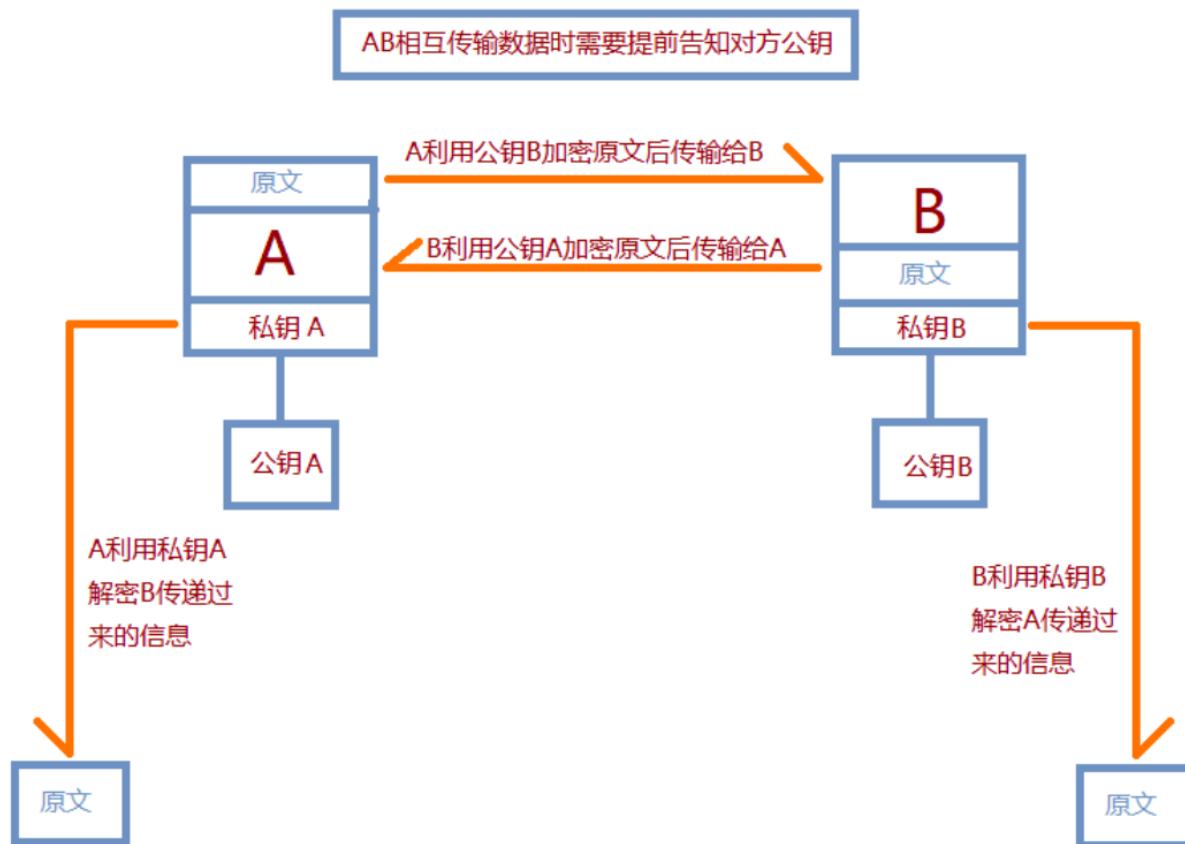
1.A要向B发送信息，A和B都要产生一对用于加密和解密的公钥和私钥。

2.A的私钥保密，A的公钥告诉B；B的私钥保密，B的公钥告诉A。

3.A要给B发送信息时，A用B的公钥加密信息，因为A知道B的公钥。

4.A将这个消息发给B（已经用B的公钥加密消息）。

5.B收到这个消息后，B用自己的私钥解密A的消息。其他所有收到这个报文的人都无法解密，因为只有B才有B的私钥。



非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对。

如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密。

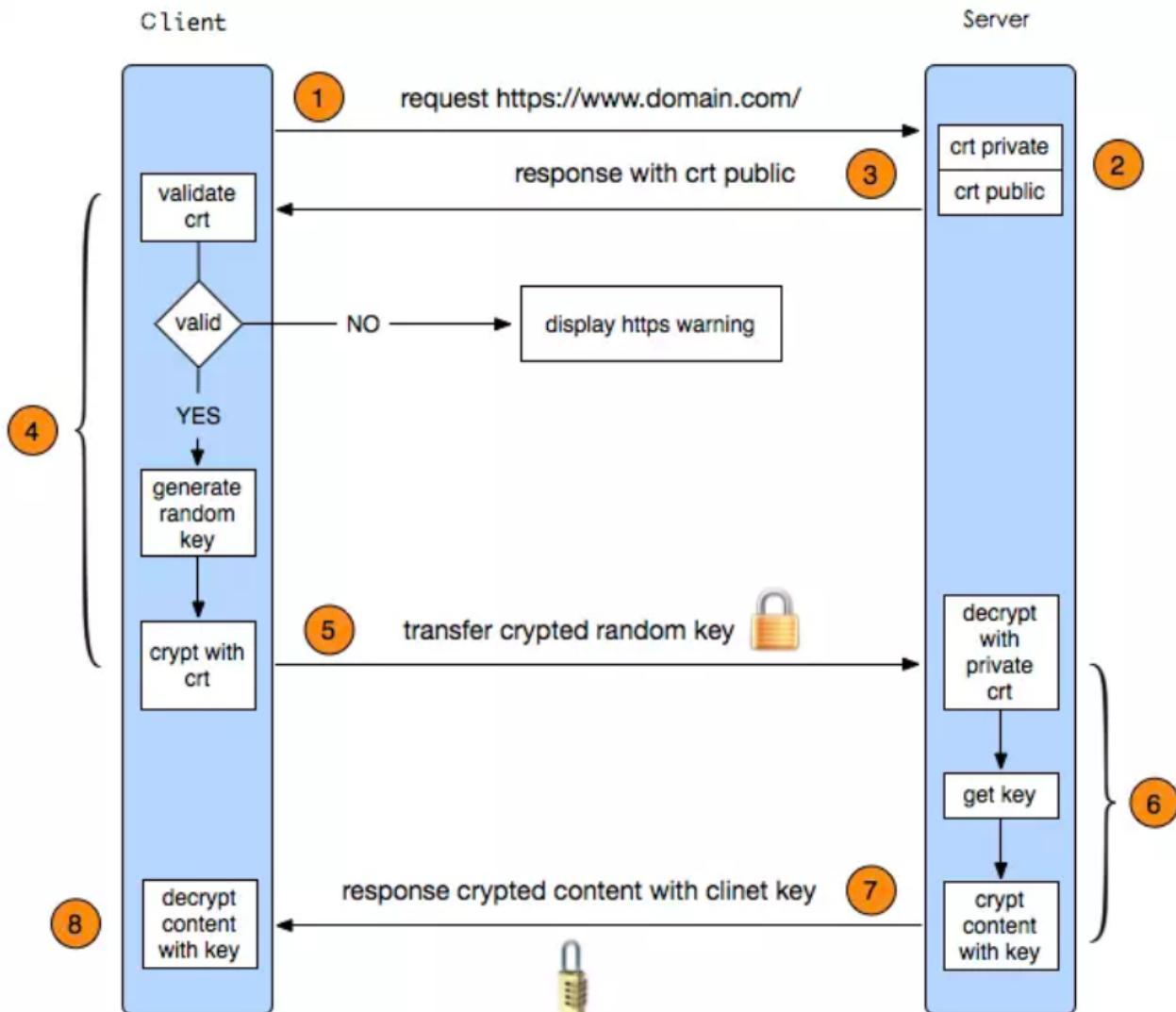
被私钥加密过的密文只能被公钥解密，过程如下：

明文 + 加密算法 + 私钥 => 密文， 密文 + 解密算法 + 公钥 => 明文

因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

私钥保密，公钥公开。

2. HTTPS请求过程



一个HTTPS请求实际上包含了两次HTTP传输，可以细分为8步。

1. 客户端向服务器发起HTTPS请求，连接到服务器的443端口
2. 服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。
3. 服务器将自己的公钥发送给客户端。
4. 客户端收到服务器端的公钥之后，会对公钥进行检查，验证其合法性，如果发现公钥有问题，那么HTTPS传输就无法继续。严格的说，这里应该是验证服务器发送的数字证书的合法性，关于客户端如何验证数字证书的合法性，下文会进行说明。如果公钥合格，那么客户端会生成一个随机值，这个随机值就是用于进行对称加密的密钥，我们将该密钥称之为client key，即客户端密钥，这样在概念上和服务器端的密钥容易进行区分。然后用服务器的公钥对客户端密钥进行非对称加密，这样客户端密钥就变成密文了，至此，HTTPS中的第一次HTTP请求结束。
5. 客户端会发起HTTPS中的第二个HTTP请求，将加密之后的客户端密钥发送给服务器。
6. 服务器接收到客户端发来的密文之后，会用自己的私钥对其进行非对称解密，解密之后的明文就是客户端密钥，然后用客户端密钥对数据进行对称加密，这样数据就变成了密文。
7. 然后服务器将加密后的密文发送给客户端。
8. 客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。这样HTTPS中的第二个HTTP请求结束，整个HTTPS传输完成。

总结：

HTTPS为了兼顾安全与效率，同时使用了对称加密和非对称加密。数据是被对称加密传输的，对称加密过程需要客户端的一个密钥，为了确保能把该密钥安全传输到服务器端，采用非对称加密对该密钥进行加密传输，总的来说，对数据进行对称加密，对称加密所要使用的密钥通过非对称加密传输。

3. HTTPS为什么不直接使用非对称加密

考虑到性能的问题：对称加密算法比非对称加密算法快大约1500倍（RSA算法生成密钥需要耗费大量时间）

6. IP地址如何划分



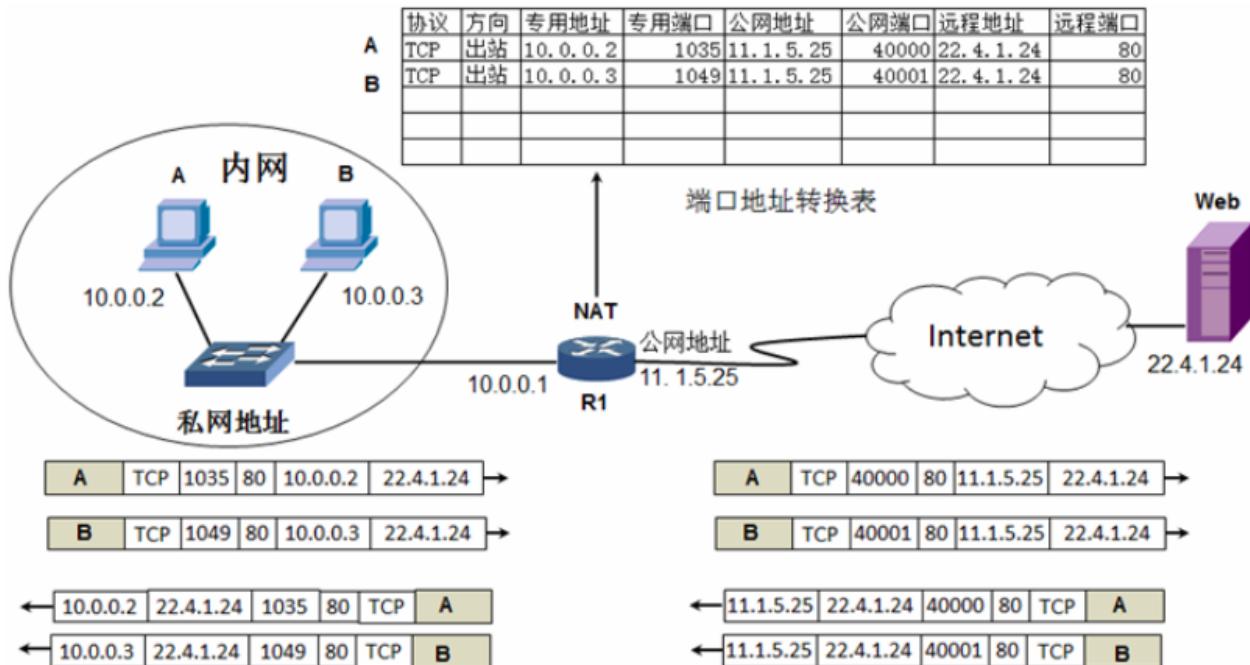
IP公网地址和私网地址？

公网IP地址 公有地址分配和管理由Inter NIC (Internet Network Information Center 因特网信息中心) 负责。各级ISP使用的公网地址都需要向Inter NIC提出申请，有Inter NIC统一发放，这样就能确保地址块不冲突。

私网IP地址 创建IP寻址方案的人也创建了私网IP地址。这些地址可以被用于私有网络，在Internet没有这些IP地址，Internet上的路由器也没有到私有网络的路由表。

- A类: 10.0.0.0 255.0.0.0, 保留了1个A类网络。
- B类: 172.16.0.0 255.255.0.0 ~ 172.31.0.0 255.255.0.0, 保留了16个B类网络。
- C类: 192.168.0.0 255.255.255.0 ~ 192.168.255.0 255.255.255.0, 保留了256个C类网络。

PS: 私网地址访问Internet需要做NAT或PAT网络地址转换



备注：NAT是什么？

NAT是 Network Address Translation 网络地址转换的缩写。NAT是将私有IP地址通过边界路由转换成外网IP地址，在边界路由的NAT地址转换表记录下这个转换映射记录，当外部数据返回时，路由使用NAT技术查询NAT转换表，再将目标地址替换成内网用户IP地址。

PAT(port address Translation, 端口地址转换，也叫端口地址复用)

这是最常用的NAT技术，也是IPv4能够维持到今天的最重要的原因之一，它提供了一种多对一的方式，对多个内网IP地址，边界路由可以给他们分配一个外网IP，利用这个外网IP的不同端口和外部进行通信。



附加问题：为什么需要使用子网掩码

虽然我们说子网掩码可以分离出IP地址中的网络部分与主机部分，可大家还是会有疑问，比如为什么要区分网络地址与主机地址？区分以后又怎样呢？那么好，让我们再详细的讲一下吧！

在使用TCP/IP协议的两台计算机之间进行通信时，我们通过将本机的子网掩码与接受方主机的IP地址进行‘与’运算，即可得到目标主机所在的网络号，又由于每台主机在配置TCP/IP协议时都设置了一个本机IP地址与子网掩码，所以可以知道本机所在的网络号。

通过比较这两个网络号，就可以知道接受方主机是否在本网络上。如果网络号相同，表明接受方在本网络上，那么可以通过相关的协议把数据包直接发送到目标主机；如果网络号不同，表明目标主机在远程网络上，那么数据包将会发送给本网络上的路由器，由路由器将数据包发送到其他网络，直至到达目的地。在这个过程中你可以看到，子网掩码是不可或缺的！

计算方式

1. 将 IP 地址与子网掩码转换成二进制；
2. 将二进制形式的 IP 地址与子网掩码做 '与' 运算，将答案化为十进制便得到网络地址；
3. 将二进制形式的子网掩码取 '反'；
4. 将取 '反' 后的子网掩码与 IP 地址做 '与' 运算，将答案化为十进制便得到主机地址。

下面我们用一个例子给大家演示：

假设有一个 IP 地址： 192.168.0.1

子网掩码为： 255.255.255.0

化为二进制为： IP 地址 11000000.10101000.00000000.00000001

子网掩码 11111111.11111111.11111111.00000000

将两者做 '与' 运算得： 11000000.10101000.00000000.00000000

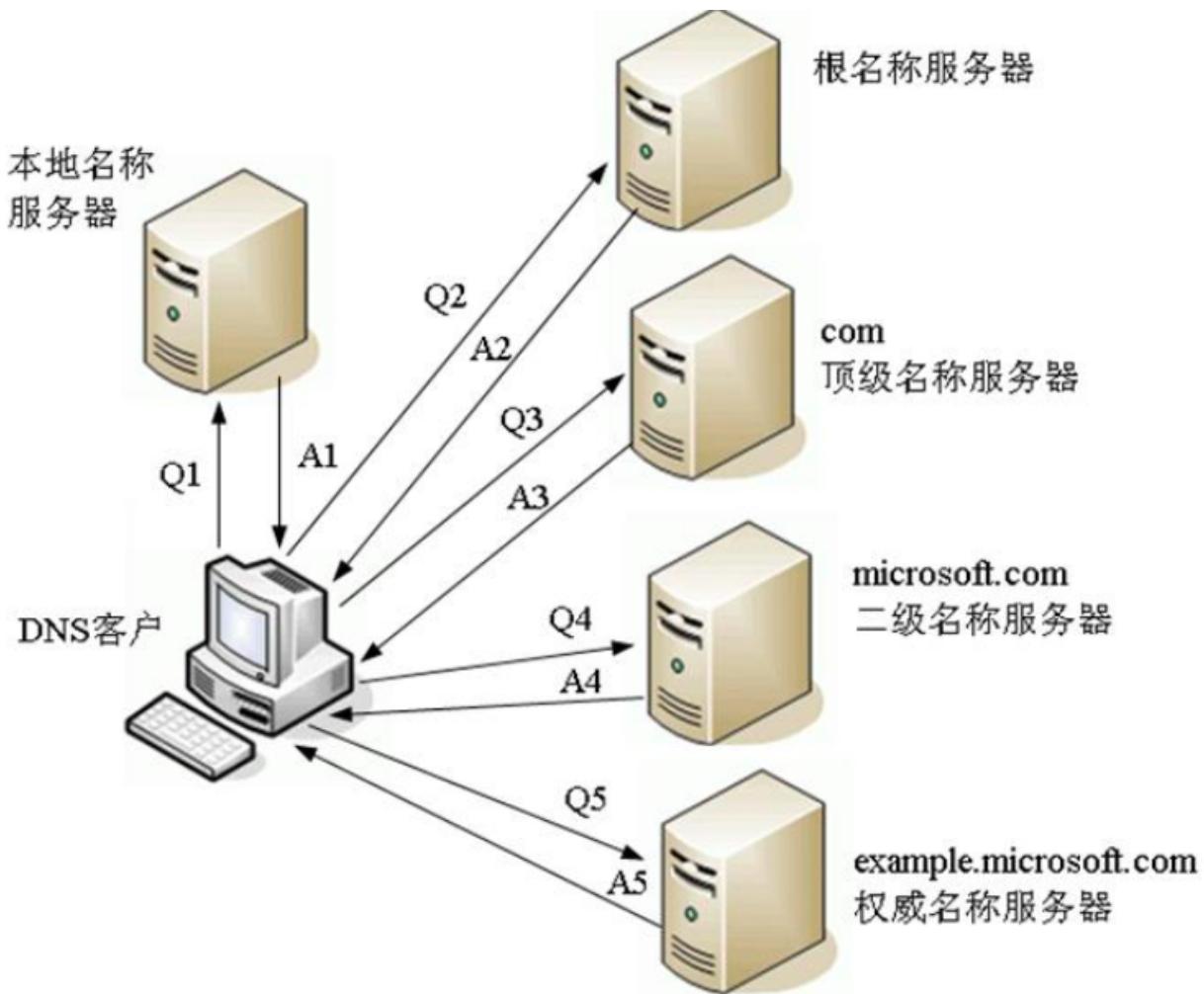
将其化为十进制得： 192.168.0.0

这便是上面 IP 的网络地址，主机地址以此类推。

小技巧：由于观察到上面的子网掩码为 C 类地址的默认子网掩码（至于为什么，可看后面的子网掩码分类就明白了），便可直接看出网络地址为 IP 地址的前三部分，即前三个字节，主机地址为最后一部分。

7. DNS解析过程 ★★

1. 查询LocalDNS
2. 查询根域名服务器(.)，得到顶级域名服务器地址
3. 查询顶级域名服务器(.com)，得到二级域名服务器地址
4. 查询二级域名服务器(baidu.com)，得到权威服务器地址
5. 查询权威服务器(www.baidu.com)，得到IP地址



8. HTTP 1.0和 1.1的区别

目前大部分使用的是HTTP 1.1协议

HTTP 1.0

HTTP 协议老的标准是HTTP/1.0，为了提高系统的效率，HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。但是，这也造成了一些性能上的缺陷，例如，一个包含有许多图像的网页文件中并没有包含真正的图像数据内容，而只是指明了这些图像的URL地址，当WEB浏览器访问这个网页文件时，浏览器首先要发出针对该网页文件的请求，当浏览器解析WEB服务器返回的该网页文档中的HTML内容时，发现其中的图像标签后，浏览器将根据标签中的src属性所指定的URL地址再次向服务器发出下载图像数据的请求。显然，访问一个包含有许多图像的网页文件的整个过程包含了多次请求和响应，每次请求和响应都需要建立一个单独的连接，每次连接只是传输一个文档和图像，上一次和下一次请求完全分离。即使图像文件都很小，但是客户端和服务器端每次建立和关闭连接却是一个相对比较费时的过程，并且会严重影响客户机和服务器的性能。当一个网页文件中包含JavaScript文件，CSS文件等内容时，也会出现类似上述的情况。

HTTP 1.1

为了克服HTTP 1.0的这个缺陷，HTTP 1.1支持持久连接（HTTP/1.1的默认模式使用带流水线的持久连接），在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。HTTP 1.1还允许客户端不用等待上一次请求结果返回，就可以发出下一

次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

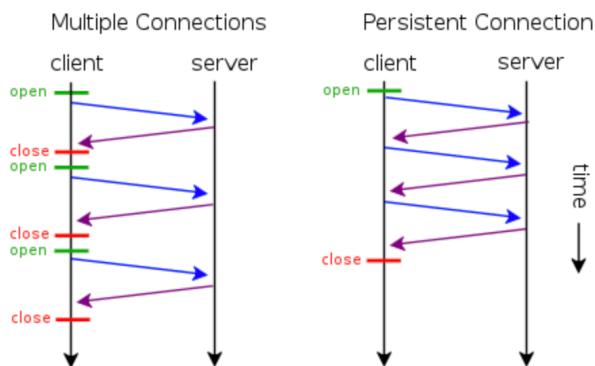
在http1.1, request和reponse头中都有可能出现一个connection的头，此header的含义是当client和server通信时对于长链接如何进行处理。在http1.1中，client和server都是默认对方支持长链接的，**如果client使用http1.1协议，但又不希望使用长链接，则需要在header中指明connection的值为close；如果server方也不想支持长链接，则在response中也需要明确说明connection的值为close。**不论request还是response的header中包含了值为close的connection，都表明当前正在使用的tcp链接在当天请求处理完毕后会被断掉。以后client再进行新的请求时就必须创建新的tcp链接了。

HTTP 1.1在继承了HTTP 1.0优点的基础上，也克服了HTTP 1.0的性能问题。HTTP 1.1通过增加更多的请求头和响应头来改进和扩充HTTP 1.0的功能。如，HTTP 1.0不支持Host请求头字段，WEB浏览器无法使用主机头名来明确表示要访问服务器上的哪个WEB站点，这样就无法使用WEB服务器在同一个IP地址和端口号上配置多个虚拟WEB站点。**在HTTP 1.1中增加Host请求头字段后，WEB浏览器可以使用主机头名来明确表示要访问服务器上的哪个WEB站点，这才实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多个虚拟WEB站点。**HTTP 1.1的持续连接，也需要增加新的请求头来帮助实现，例如，Connection请求头的值为Keep-Alive时，客户端通知服务器返回本次请求结果后保持连接；Connection请求头的值为close时，客户端通知服务器返回本次请求结果后关闭连接。HTTP 1.1还提供了与身份认证、状态管理和Cache缓存等机制相关的请求头和响应头。HTTP/1.0不支持文件断点续传，`RANGE:bytes`是HTTP/1.1新增内容，HTTP/1.0每次传送文件都是从文件头开始，即0字节处开始。`RANGE:bytes=xxxx`表示要求服务器从文件XXXX字节处开始传送，这就是我们平时所说的断点续传！

- HTTP 1.1 增加了Host字段，可以判定访问哪个主机
- HTTP 1.1 默认开启了Keep-Alive模式，可以减少TCP连接建立的消耗

1. 什么是Keep-Alive模式

我们知道HTTP协议采用“请求-应答”模式，当使用普通模式，即非KeepAlive模式时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（HTTP协议为无连接的协议）；当使用Keep-Alive模式（又称持久连接、连接重用）时，Keep-Alive功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive功能避免了建立或者重新建立连接。



http 1.0中默认是关闭的，需要在http头加入"Connection: Keep-Alive"，才能启用Keep-Alive；http 1.1中默认启用Keep-Alive，如果加入"Connection: close "，才关闭。目前大部分浏览器都是用http1.1协议，也就是说默认都会发起Keep-Alive的连接请求了，所以是否能完成一个完整的Keep-Alive连接就看服务器设置情况。

2. 常见的HTTP请求头

请求头	说明
Host	接受请求的服务器地址，可以是IP端口号，也可以是域名
User-Agent	发送请求的应用程序名称
Connection	指定与连接相关的属性，如Connection:Keep-Alive
Accept-Charset	通知服务端可以发送的编码格式
Accept-Encoding	通知服务端可以发送的数据压缩格式
Accept-Language	通知服务端可以发送的语言

9. Restful API

就是用URL定位资源，用HTTP描述操作。

看Url就知道要什么

看http method就知道干什么

看http status code就知道结果如何

10. HTTP常用方法

1. POST和PUT区别

POST请求会向指定资源提交数据，请求服务器进行处理，如：表单数据提交、文件上传等，请求数据会被包含在请求体中。**POST方法是非幂等的方法**，因为这个请求可能会创建新的资源或/和修改现有资源。

PUT请求会身向指定资源位置上传其最新内容，**PUT方法是幂等的方法**。通过该方法客户端可以将指定资源的最新数据传送给服务器取代指定的资源的内容。

2. GET和HEAD区别

HEAD方法与GET方法一样，都是向服务器发出指定资源的请求。但是，服务器在响应HEAD请求时不会回传资源的内容部分，即：响应主体。这样，我们可以不传输全部内容的情况下，就可以获取服务器的响应头信息。**HEAD方法常被用于客户端查看服务器的性能**。

3. OPTIONS

OPTIONS请求与HEAD类似，一般也是用于客户端查看服务器的性能。这个方法会请求服务器返回该资源所支持的所有HTTP请求方法，该方法会用'*'来代替资源名称，向服务器发送OPTIONS请求，可以测试服务器功能是否正常。JavaScript的XMLHttpRequest对象进行CORS跨域资源共享时，就是使用OPTIONS方法发送嗅探请求，以判断是否有对指定资源的访问权限。

4. TRACE

TRACE请求服务器回显其收到的请求信息，该方法主要用于HTTP请求的测试或诊断。

5. PATCH

PATCH方法出现的较晚，它在2010年的RFC 5789标准中被定义。PATCH请求与PUT请求类似，同样用于资源的更新。二者有以下两点不同：**1.PATCH一般用于资源的部分更新，而PUT一般用于资源的整体更新。****2.当资源不存在时，PATCH会创建一个新的资源，而PUT只会对已在资源进行更新。**

11. 常用协议以及端口

协议	端口	传输层
DNS	53	UDP
HTTP	80	TCP
HTTPS	443	TCP
DHCP	67	UDP
FTP	默认情况下FTP协议使用TCP端口中的 20和21这两个端口，其中20用于传输数据，21用于传输控制信息。	TCP

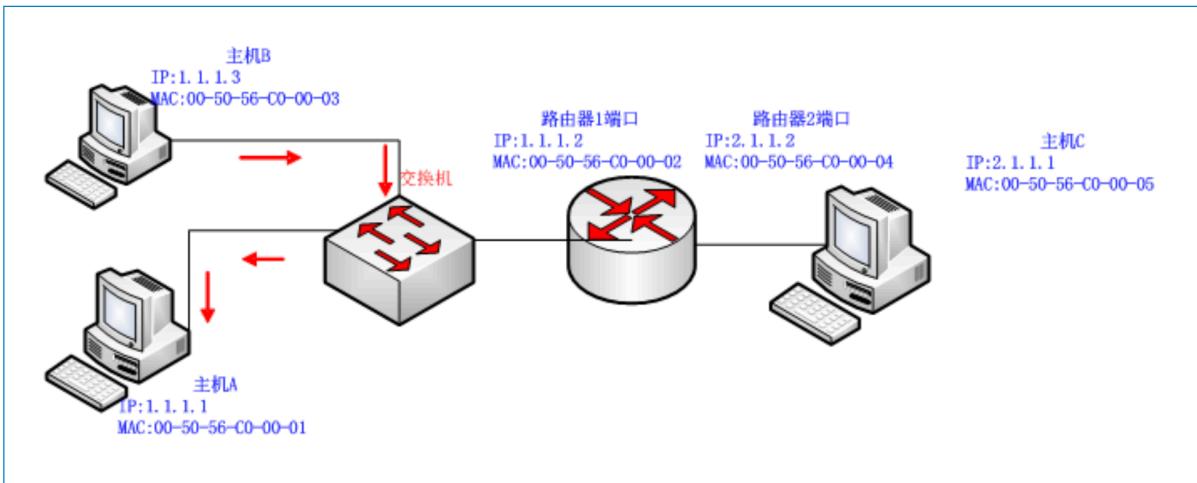
003 网络层协议

1. PING原理

- 作用：检测两台计算机是否联通
- 核心是ICMP协议
- 过程
 - 首先，Ping命令会构建一个固定格式的ICMP请求数据包，然后由ICMP协议将这个数据包连同地址“192.168.1.2”一起交给IP层协议（和ICMP一样，实际上是一组后台运行的进程），IP层协议将以地址“192.168.1.2”作为目的地址，本机IP地址作为源地址，加上一些其他的控制信息，构建一个IP数据包，并在一个映射表中查找出IP地址192.168.1.2所对应的物理地址

(也叫MAC地址，熟悉网卡配置的朋友不会陌生，这是数据链路层协议构建数据链路层的传输单元——帧所必需的），一并交给数据链路层。后者构建一个数据帧，目的地址是IP层传过来的物理地址，源地址则是本机的物理地址，还要附加上一些控制信息，依据以太网的介质访问规则，将它们传送出去。其中映射表由ARP实现。ARP(Address Resolution Protocol)是地址解析协议,是一种将IP地址转化成物理地址的协议。**ARP**具体说来就是将网络层（IP层，也就是相当于OSI的第三层）地址解析为数据连接层（MAC层，也就是相当于OSI的第二层）的MAC地址。

- 主机B收到这个数据帧后，先检查它的目的地址，并和本机的物理地址对比，如符合，则接收；否则丢弃。接收后检查该数据帧，将IP数据包从帧中提取出来，交给本机的IP层协议。同样，IP层检查后，将有用的信息提取后交给ICMP协议，后者处理后，马上构建一个ICMP应答包，发送给主机A，其过程和主机A发送ICMP请求包到主机B一模一样。



1.ICMP协议

前面讲到了，IP协议并不是一个可靠的协议，它不保证数据被送达，那么，自然的，保证数据送达的工作应该由其他的模块来完成。其中一个重要的模块就是ICMP(网络控制报文)协议。

当传送IP数据包发生错误——比如主机不可达，路由不可达等等，ICMP协议将会把错误信息封包，然后传送给主机。给主机一个处理错误的机会，这也就是说建立在IP层以上的协议是可能做到安全的原因。

004 链路层

0x5 Coding

001 数据结构 | 算法

主要考察排序、链表、二叉树。

- 时间复杂度定义

- 一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 $T(n)$ 表示，若有某个辅助函数 $f(n)$ ，使得 $T(n)/f(n)$ 的极限值（当 n 趋于无穷大时）为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。

排序法	平均时间	最差情形	稳定性	额外空间	备注
冒泡	$O(n^2)$	$O(n^2)$	稳定	$O(1)$	n 小时较好
交换	$O(n^2)$	$O(n^2)$	不稳定	$O(1)$	n 小时较好
选择	$O(n^2)$	$O(n^2)$	不稳定	$O(1)$	n 小时较好
插入	$O(n^2)$	$O(n^2)$	稳定	$O(1)$	大部分已排序时较好
基数	$O(\log RB)$	$O(\log RB)$	稳定	$O(n)$	B是真数(0-9), R是基数(个十百)
Shell	$O(n \log n)$	$O(ns)$ $1 < s < 2$	不稳定	$O(1)$	s 是所选分组
快速	$O(n \log n)$	$O(n^2)$	不稳定	$O(n \log n)$	n 大时较好
归并	$O(n \log n)$	$O(n \log n)$	稳定	$O(1)$	n 大时较好
堆	$O(n \log n)$	$O(n \log n)$	不稳定	$O(1)$	n 大时较好

1. 堆排

```

package _01_Algorithm.Sort;

import java.util.Arrays;

/**
 * 大顶堆-->升序排列
 *
 * http://www.cnblogs.com/chengxiao/p/6129630.html
 * (1) 首先要定义大顶堆: arr[0...n-1] 则 arr[i] > arr[2*i+1] 并且 arr[i] >
arr[2*i+2]
 * (2) 对于最开始的数组生成一个大顶堆: 从最后一个非叶子节点开始调整
 * (3) 对于生成的大顶堆, 根节点元素是整个数组中最大的元素
 *      (i) 首先将根节点元素和数组最后一个元素置换, 则最后一个元素是最大的元素 【有一点冒泡
排序的处理味道】
 *          (ii) 上面的操作会打乱大顶堆, 所以需要对于arr[0...n-2]重新生成大顶堆
 * (4) 堆排序中涉及到的树的概念, 但是并没有显示的存储树, 而是根据数组元素位置和树节点之间的
对应关系调整假想中的树, 有点意思
 *
 * 堆排序可以用来解决topK问题: 从100w个数中找到最大的k个数
 * (1) 首先对于数组前k个数建立一个小顶堆
 * (2) 然后从第k个数到最后一个数, 如果数比堆顶元素大, 那么交换

```

```

* (3) 调整小顶堆的顶部元素
* (4) 最后得到的小顶堆就是从小到大的元素
*/
public class HeapSort {

    public static void heapSort(int[] arr){
        //0. 首先是建立一个大顶堆
        for(int i=arr.length/2-1; i >=0 ; i--){ //1. 从第一个非叶子节点开始，直到顶端
            root节点
            adjustHeap(arr,i,arr.length);
        }

        //1. 然后不断交换元素，缩小大顶堆
        for(int j=arr.length-1;j > 0; j--){ //2. 将大顶堆第一个元素和最后一个元素替换，替换之后重新调整堆
            swap(arr,0,j);
            adjustHeap(arr,0,j); //这里只需要调整被打乱的根节点即可
        }
    }

    //3. 参数：数组本身，调整节点的下标，数组的长度
    public static void adjustHeap(int[] arr, int i, int length){ //i的含义：调整
树中第i个节点【和左右子树对比】
        int temp = arr[i];
        for(int k = 2*i+1; k < length; k = 2*k+1){
            if(k+1 < length && arr[k] < arr[k+1]) //找到左右子树中最大的【反证：如果
左子树<右子树，那么交换之后根节点还是小于右子树】
                k += 1;

            if(arr[k] > temp){ //对比子树和父节点
                arr[i] = arr[k];
                i = k; //注意为什么每次要改变i的值
            }else{
                break; //不需要调整，完事
            }
        }
        arr[i] = temp; //归位
    }

    public static void swap(int[] arr, int a, int b){
        int temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }

    public static void main(String[] args){
        int[] arr = {9,8,7,6,5,4,3,2,1};
        heapSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}

```

```
    }  
}
```

2. 快排

```
public static void quickSort(int[] nums, int left, int right){  
    int i = left, j = right;  
    int base = nums[left];  
    while(i < j){  
        while(nums[j] >= base && i < j)  
            j--;  
        while(nums[i] <= base && i < j)  
            i++;  
        if(i < j){  
            int temp = nums[i];  
            nums[i] = nums[j];  
            nums[j] = temp;  
        }  
    }  
    nums[left] = nums[i];  
    nums[i] = base;  
    if(left < i-1) {  
        quickSort(nums, left, i - 1);  
    }  
    if(i+1 < right) {  
        quickSort(nums, i + 1, right);  
    }  
}
```

关于排序稳定性的定义

通俗地讲就是能保证排序前两个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。在简单形式化一下，如果 $A_i = A_j$, A_i 原来在位置前，排序后 A_i 还是要在 A_j 位置前。

1. 快速排序是否是稳定排序

不稳定。比如序列为 5 3 3 4 3 8 9 10 11，现在中枢元素 5 和 3（第 5 个元素，下标从 1 开始计）交换就会把元素 3 的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和 $a[j]$ 交换的时刻。

2. 哪个常见排序是稳定的，为什么

冒泡排序，两个元素如果相同，可以控制不用发生交换。

3. 跳表

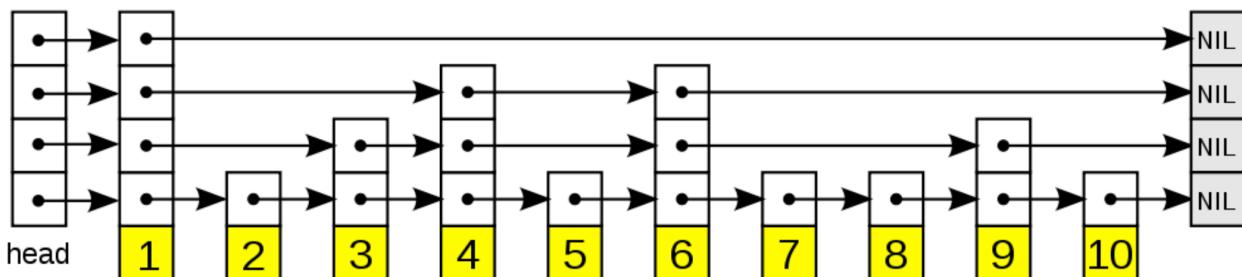
Skip list(跳表) 是一种可以代替平衡树的数据结构，默认是按照Key值升序的。Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过“空间来换取时间”的一个算法，在每个节点中增加了向前的指针，在插入、删除、查找时可以忽略一些不可能涉及到的结点，从而提高了效率。

在Java的API中已经有了实现：分别是ConcurrentSkipListMap(在功能上对应HashTable、HashMap、TreeMap)；ConcurrentSkipListSet(在功能上对应HashSet)

跳跃表以有序的方式在层次化的链表中保存元素，效率和AVL树媲美——查找、删除、添加等操作都可以在O(LogN)时间下完成，并且比起二叉搜索树来说，跳跃表的实现要简单直观得多。

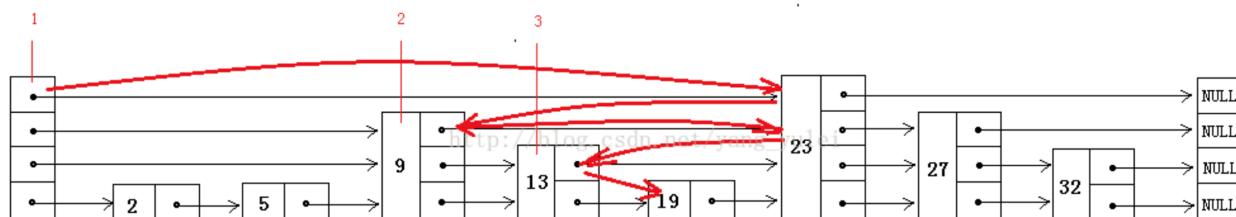
跳表的构造过程是：

- 1、给定一个有序的链表。
- 2、选择连表中最大和最小的元素，然后从其他元素中按照一定算法随即选出一些元素，将这些元素组成有序链表。这个新的链表称为一层，原链表称为其下一层。
- 3、为刚选出的每个元素添加一个指针域，这个指针指向下一层中值同自己相等的元素。Top指针指向该层首元素
- 4、重复2、3步，直到不再能选择出除最大最小元素以外的元素。



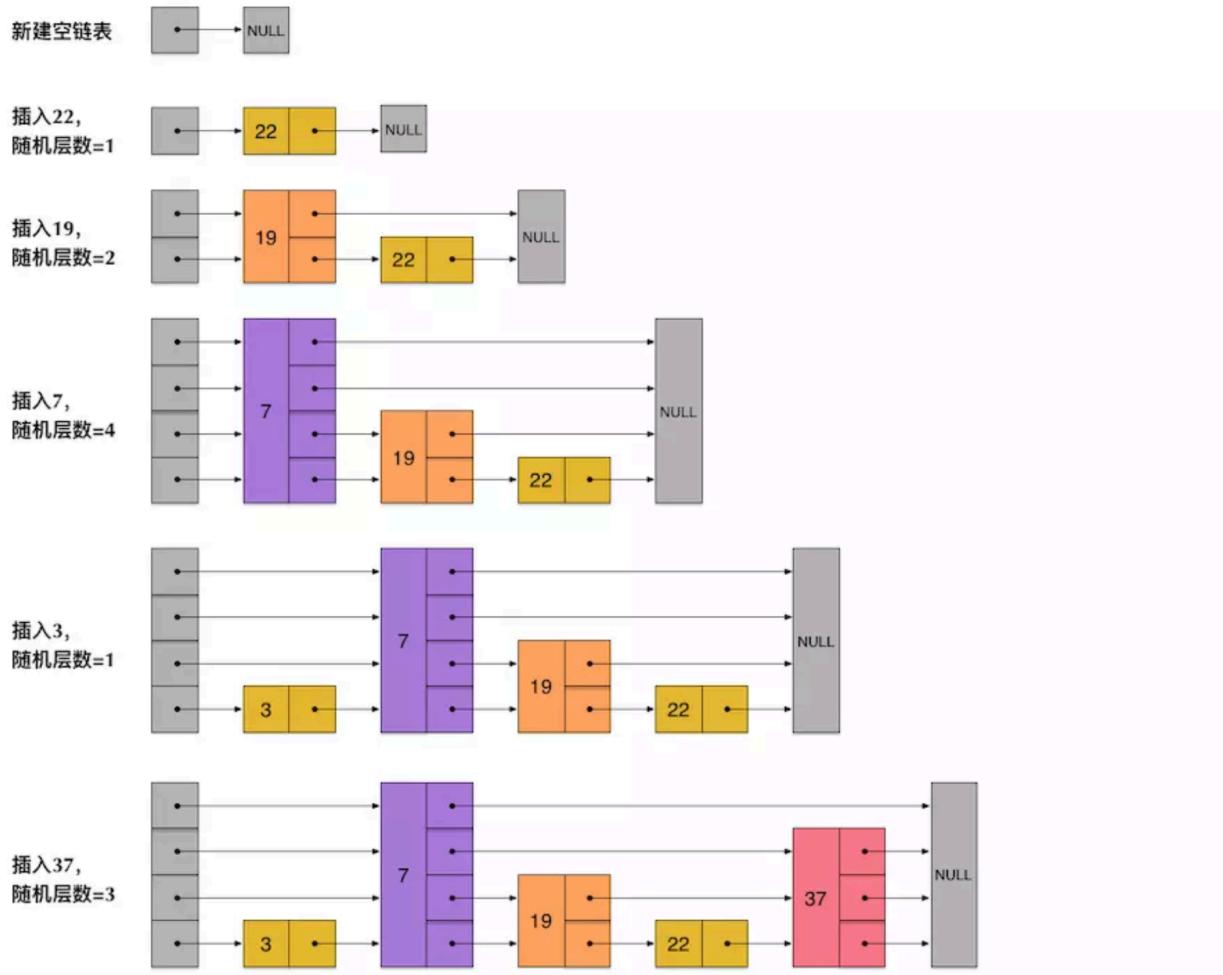
1. 查找

从最上层往下查找，平均时间复杂度O(logN)



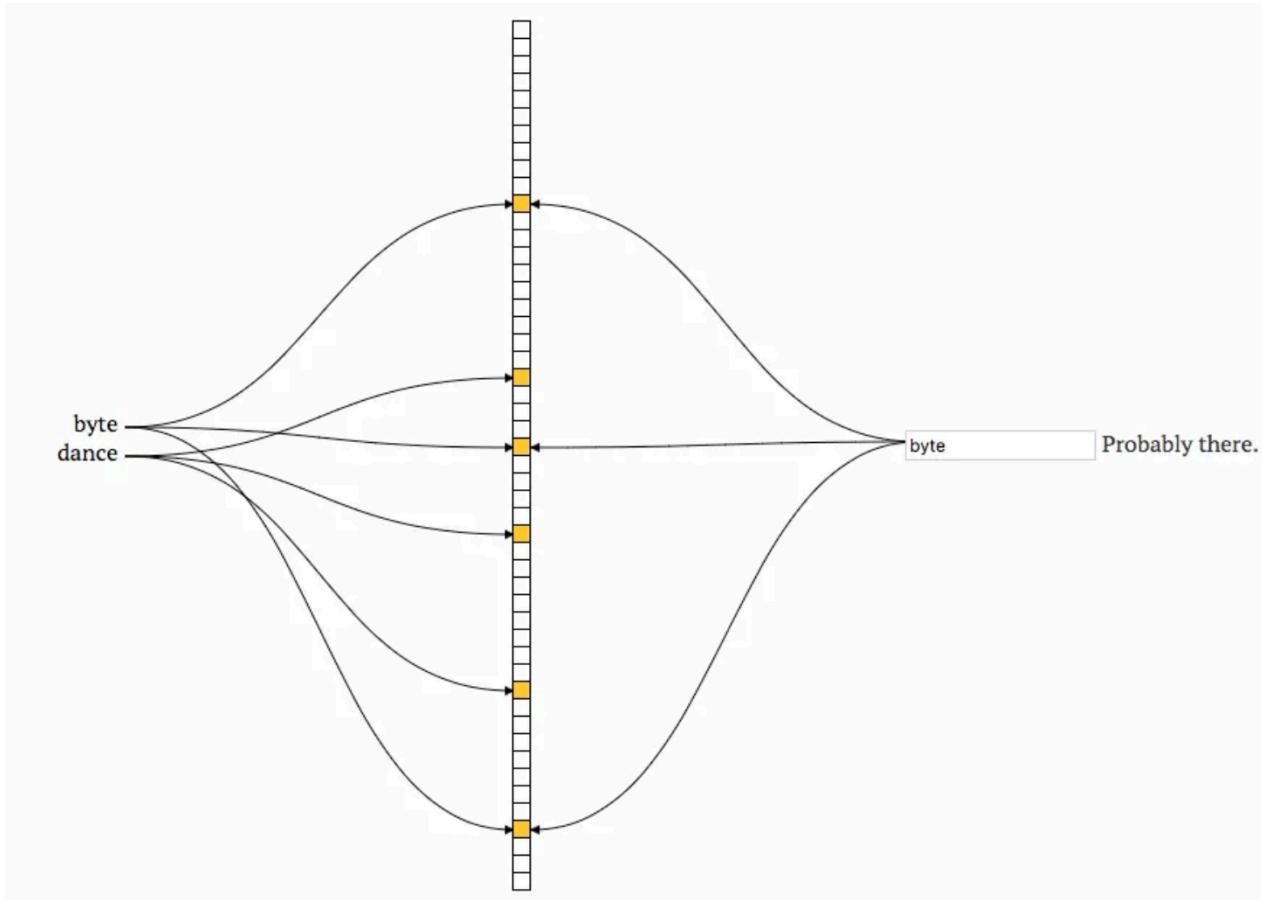
2. 插入

- 为什么要随机节点出现的次数
 - 新插入一个节点之后，就会打乱上下相邻两层链表上节点个数严格的2:1的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点（也包括新插入的节点）重新进行调整，这会让时间复杂度重新蜕化成O(n)。删除数据也有同样的问题。
 - skipList为了避免这一问题，它不要求上下相邻两层链表之间的节点个数有严格的对应关系，而是为每个节点随机出一个层数(level)。比如，一个节点随机出的层数是3，那么就把它链入到第1层到第3层这三层链表中。



4. 布隆过滤器

<https://zhuanlan.zhihu.com/p/43263751>



- 能够判定元素一定不存在
- 能够判定元素可能存在

(1)添加元素

布隆过滤器的原理是，当一个元素被加入集合时，通过K个hash函数将这个元素映射成一个位数组中的K个点，把它们置为1。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是布隆过滤器的基本思想。

<因为不同的key，可能经过K个hash函数之后产生的多个位下标是相同的[或者多个key构成了一个新的映射集合]，导致误判>

(2)判断元素是否存在

当我们判断一个元素是否在布隆过滤器中时，我们把这个值传入k个hash函数中获得映射的k个点。这一次我们确认下是否所有的点都被置为1了，如果有某一位没有置为1则这个元素肯定不在集合中。如果都在那这个元素就有可能在集合中。

5. B树/B+树

B树和B+树的时间复杂度如下：

算法	平均	最差
空间	$O(n)$	$O(n)$
搜索	$O(\log n)$	$O(\log n)$
插入	$O(\log n)$	$O(\log n)$
删除	$O(\log n)$	$O(\log n)$

6. 二叉树遍历の非递归写法

前序遍历

```
//1. 递归
private static List<Integer> res;
public void PreOrder(TreeNode root){
    if(root != null){ //递归终止条件
        res.add(root.val);
        PreOrder(root.left); //类似入栈
        preOrder(root.right);
    }
    //程序自动出栈
}

//2. 非递归
public void PreOrder(TreeNode root){
    List<Integer> res = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    TreeNode cur = root;

    while(cur != null || !stack.isEmpty()){
        while(cur != null){
            stack.push(cur);
            res.add(cur.val);
            cur = cur.left;
        }
        //这里cur = null, 等同于递归终止条件
        cur = stack.pop(); //刚pop的元素已经使用过了,
        cur = cur.right; //所以需要cur.right
    }
}
```

中序遍历

```

public void InOrder(TreeNode root){
    List<Integer> res = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();

    TreeNode cur = root;
    while(cur != null || !stack.isEmpty()){
        if(cur != null){
            stack.push(cur);
            cur = cur.left;
        }
        cur = stack.pop();
        res.add(cur.val); //在这里才访问节点
        cur = cur.right;
    }
}

```

后序遍历

后序遍历的非递归实现难度最高

//1.递归实现

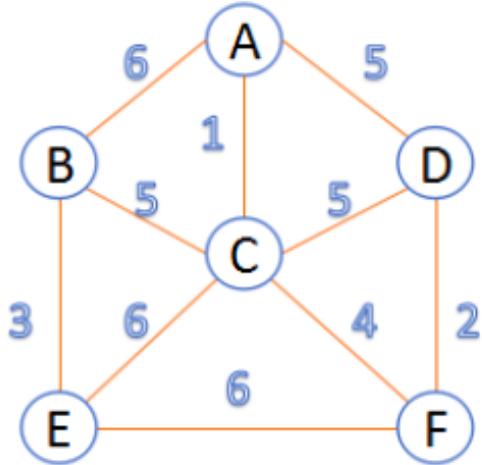
//2.非递归实现

7. 最小生成树【prim算法】

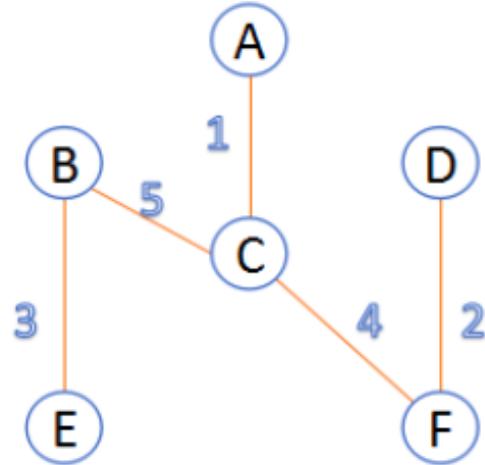
加权联通图中找到权值和最小的生成树

1. 基本定义

- **连通图**: 在无向图中, 若任意两个顶点 v_i 与 v_j 都有路径相通, 则称该无向图为连通图。
- **强连通图**: 在有向图中, 若任意两个顶点 v_i 与 v_j 都有路径相通, 则称该有向图为强连通图。
- **连通网**: 在连通图中, 若图的边具有一定的意义, 每一条边都对应着一个数, 称为权; 权代表着连接个顶点的代价, 称这种连通图叫做连通网。
- **生成树**: 一个连通图的生成树是指一个连通子图, 它含有图中全部 n 个顶点, 但只有足以构成一棵树的 $n-1$ 条边。一颗有 n 个顶点的生成树有且仅有 $n-1$ 条边, 如果生成树中再添加一条边, 则必定成环。
- **最小生成树**: 在连通网的所有生成树中, 所有边的代价和最小的生成树, 称为最小生成树。

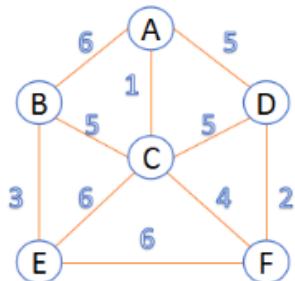


连通网G



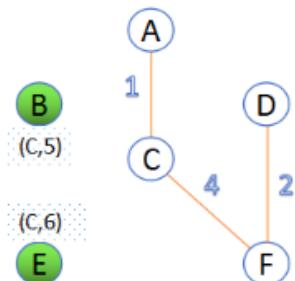
最小生成树

2. 查找过程

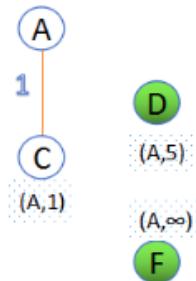


连通网G

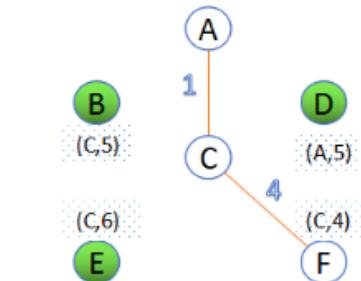
1. 初始 $u=\{A\}, v=\{B, C, D, E, F\}$; 顶点 B 下方 $(A, 6)$, 表示与集合 u 中 A 的代价为 6 作为最小代价边。选择最小的代价边 (A, C) , 把 C 并入到集合 u 中。



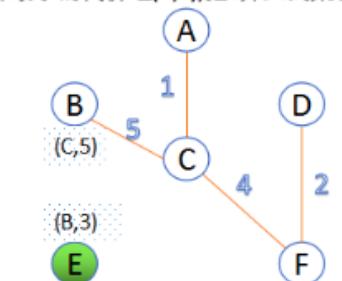
3. $u=\{A, C, F\}, v=\{B, D, E\}$; 更新 v 中顶点与集合 u 的最小的代价边; 选择最小代价边 (F, D) , D 并入 u 。



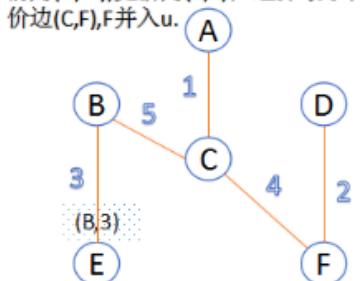
1. 初始 $u=\{A\}, v=\{B, C, D, E, F\}$; 顶点 B 下方 $(A, 6)$, 表示与集合 u 中 A 的代价为 6 作为最小代价边。选择最小的代价边 (A, C) , 把 C 并入到集合 u 中。



2. $u=\{A, C\}, v=\{B, D, E, F\}$; 更新 v 中顶点与集合 u 的最小的代价边; 例如: 顶点 E 之前为 (A, ∞) , 更新为 $(C, 6)$; 选择最小代价边 (C, F) , F 并入 u 。



4. $u=\{A, C, F, D\}, v=\{B, E\}$; 更新 v 中顶点与集合 u 的最小的代价边; 选择最小代价边 (C, B) , B 并入 u 。

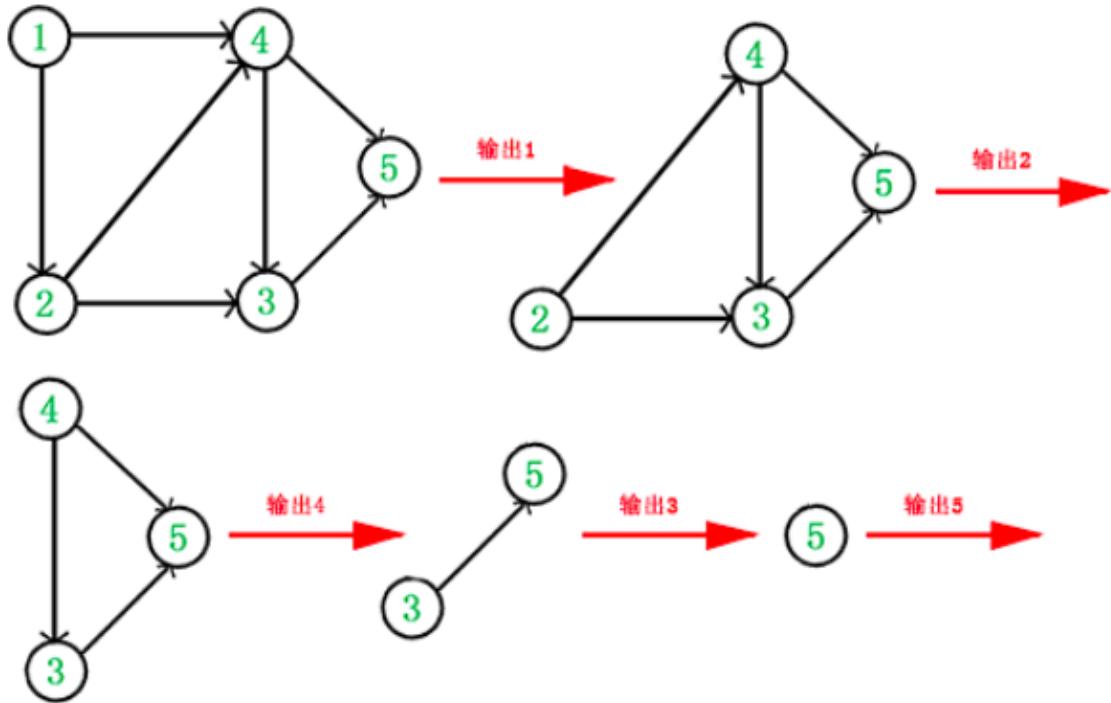


5. $u=\{A, C, F, D, B\}, v=\{E\}$; 更新 v 中顶点与集合 u 的最小的代价边; 选择最小代价边 (B, E) , E 并入 u 。

8. 拓扑排序

- 在图论中, 拓扑排序 (Topological Sorting) 是一个有向无环图 (DAG, Directed Acyclic Graph) 的所有顶点的线性序列。且该序列必须满足下面两个条件:
 - 每个顶点出现且只出现一次。
 - 若存在一条从顶点 A 到顶点 B 的路径, 那么在序列中顶点 A 出现在顶点 B 的前面。

- 从 DAG 图中选择一个没有前驱 (即入度为 0) 的顶点并输出。
- 从图中删除该顶点和所有以它为起点的有向边。
- 重复 1 和 2 直到当前的 DAG 图为空或当前图中不存在无前驱的顶点为止。后一种情况说明有向图中必然存在环。



9. 并查集

TODO

<https://www.cnblogs.com/hapjin/p/5478352.html>

10. 二叉树

1. 完全二叉树和平衡二叉树的区别

平衡二叉树【AVL树】是BST的一种优化。

AVL树，是一种平衡(balanced)的二叉搜索树(binary search tree, 简称为BST)。由两位科学家在1962年发表的论文《An algorithm for the organization of information》当中提出，作者是发明者[G.M. Adelson-Velsky](#)和[E.M. Landis](#)（链接由维基百科提供）。它具有以下两个性质：

- 任意一个结点的key，比它的左孩子key大，比它的右孩子key小；【同时具备二叉查找树的性质】
- 任意结点的孩子结点之间高度差距最大为1；【为什么平衡】

2. 完全二叉树和满二叉树的区别

若设二叉树的深度为 h ，除第 h 层外，其它各层 ($1 \sim h-1$) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

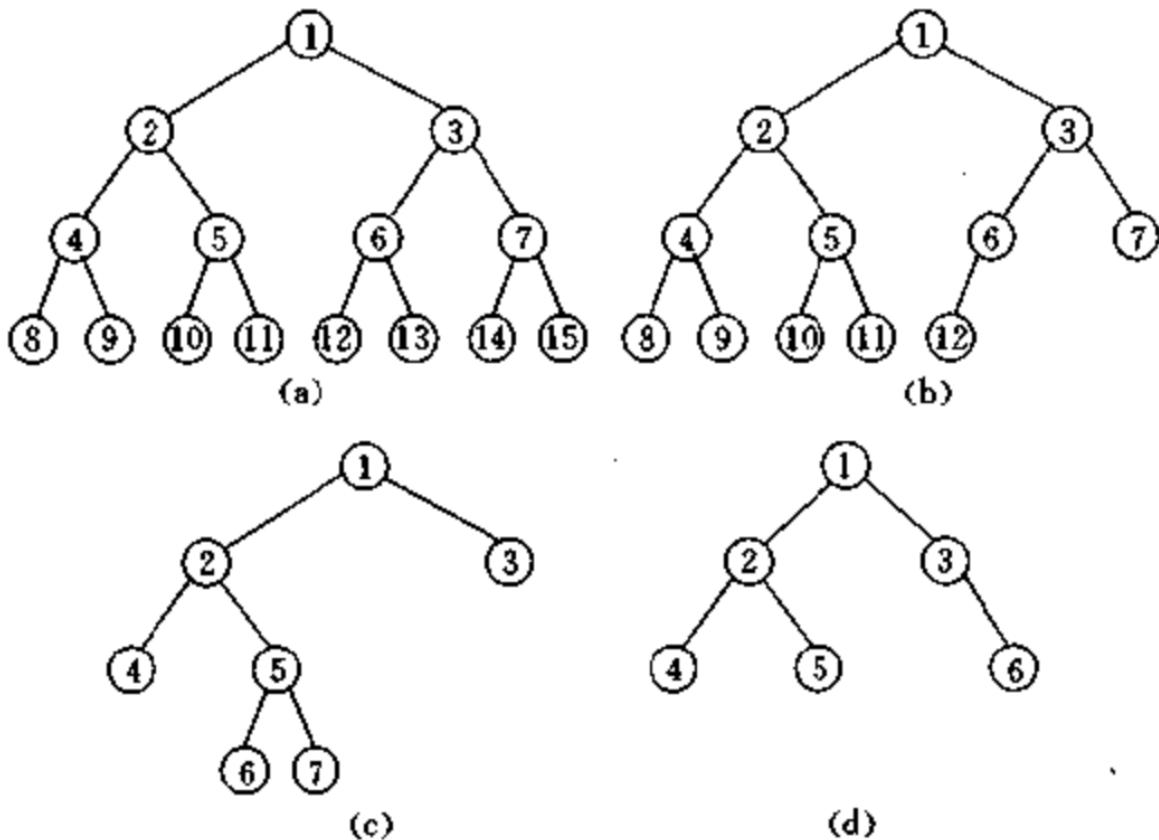


图 6.4 特殊形态的二叉树

(a) 满二叉树; (b) 完全二叉树; (c)和(d)非完全二叉树。

3. 求解完全二叉树的深度

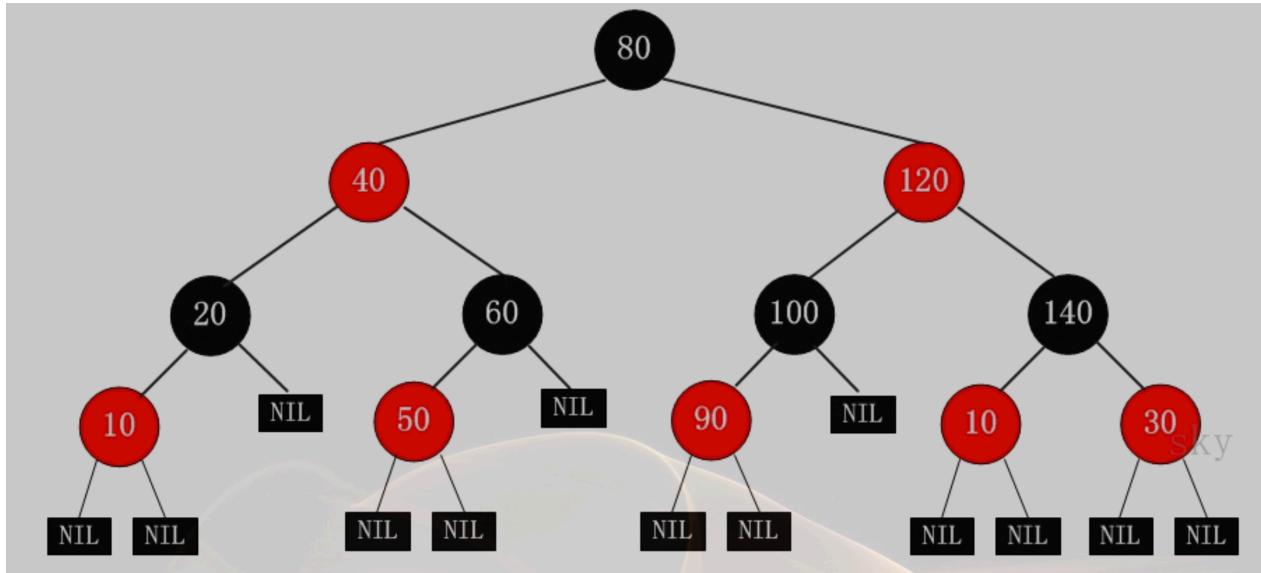
因为完全二叉树最底层从左到右是连续的，所以左边走到底就能获得最大高度【深度】。

11. 红黑树

1. 红黑树定义

红黑树是一种二叉查找树【BST树的强化：有自平衡特性】

- 性质1：每个节点要么是黑色，要么是红色。
- 性质2：根节点是黑色。
- 性质3：每个叶子节点（NIL）是黑色。
- 性质4：每个红色结点的两个子结点一定都是黑色。
- 性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。



2.

002 LeetCode

一维DP

二维DP

DFS

剪枝

0x6 数据库

001 MySQL

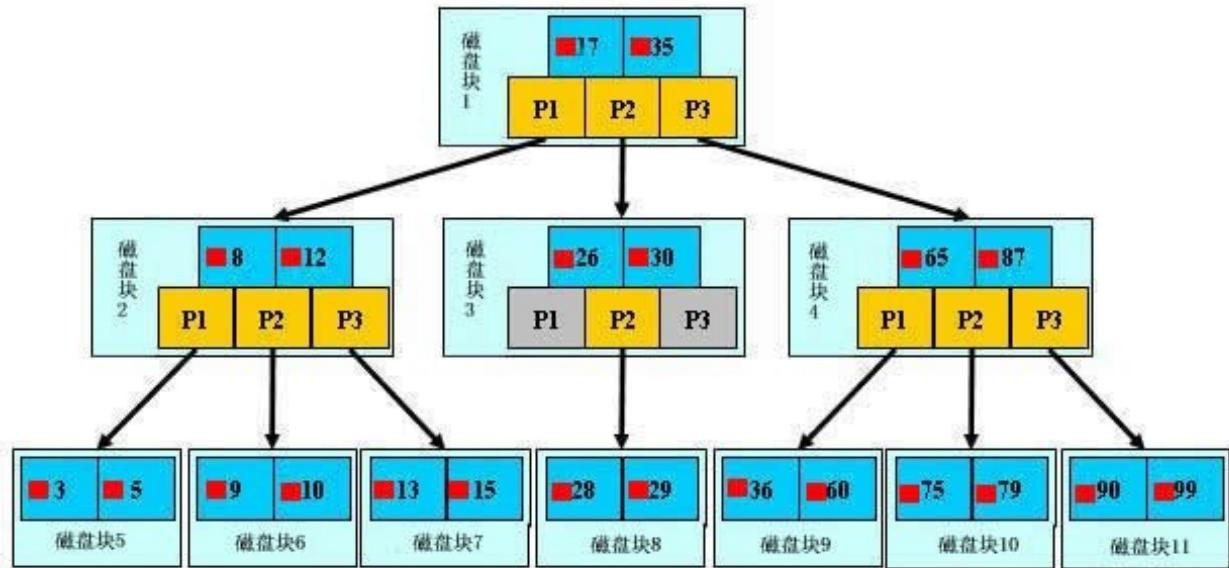
1. 索引 ★★★★★

1. 底层数据结构

0. B+树

MySQL数据库索引使用B+树：

B+树的特点：B+树是应文件系统所需而产生的一种B树的变形树（文件的目录一级一级索引，只有最底层的叶子节点（文件）保存数据）非叶子节点只保存索引，不保存实际的数据，数据都保存在叶子节点中。



浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

查询过程：

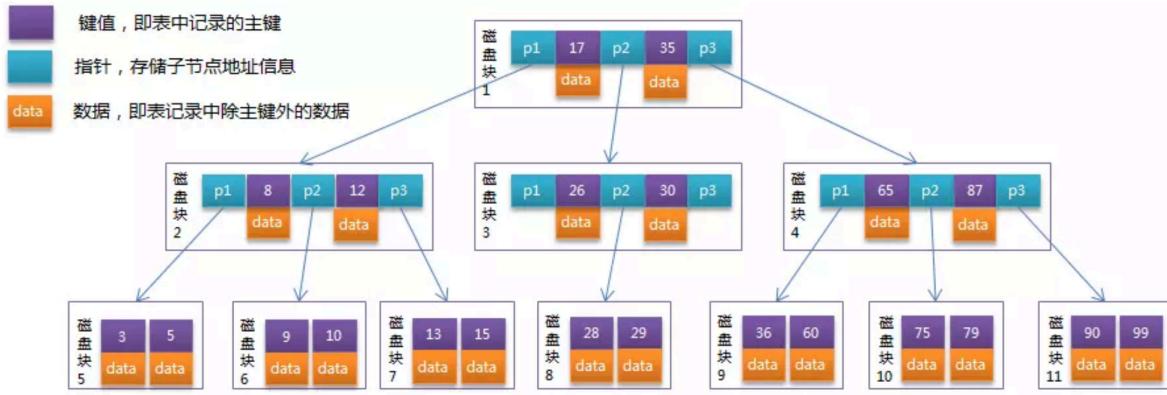
如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的B+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常高。

1. 相对于二叉查找树、AVL树，为什么要使用B/B+树

- AVL树和红黑树基本都是存储在内存中才会使用的数据结构。在大规模数据存储的时候，红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率底下的情况。
 - 红黑树、AVL树都是二叉树，存放相同的数据深度更大。

2. 为什么要使用B+树不使用B树

首先要了解B树：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

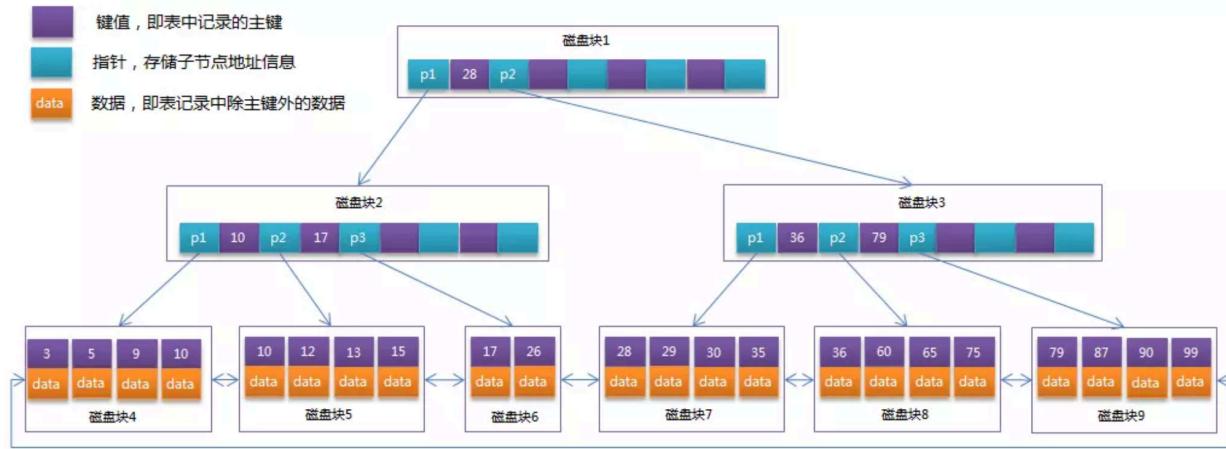
模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间 (17,35)，找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间 (26,30)，找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。

然后了解为什么有B+树：

- B+树是B树的优化
 - 更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。
 - 从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。
- 总结一下
 - 效率方面：如果data很大，磁盘空间有限，B树的非叶子节点能够存放的key比较少—>对于同样的数据量，每个节点的子树更少，导致深度加大—>可能增加磁盘I/O次数
 - 区间访问性能：数据都在叶子节点上，并且增加了顺序访问指针，每个叶子节点都指向相邻的叶子节点的地址。相比B-Tree来说，进行范围查找时只需要查找两个节点，进行遍历即可，提高了区间访问性能（无需返回上层父节点重复遍历查找减少IO操作）。而B-Tree需要获取所有节点，相比之下B+Tree效率更高。



- 相对于B树，B+树：

- 数据都存放在叶子节点
- 底层叶子节点有顺序指针

3. 为什么要使用B+树不使用Hash表

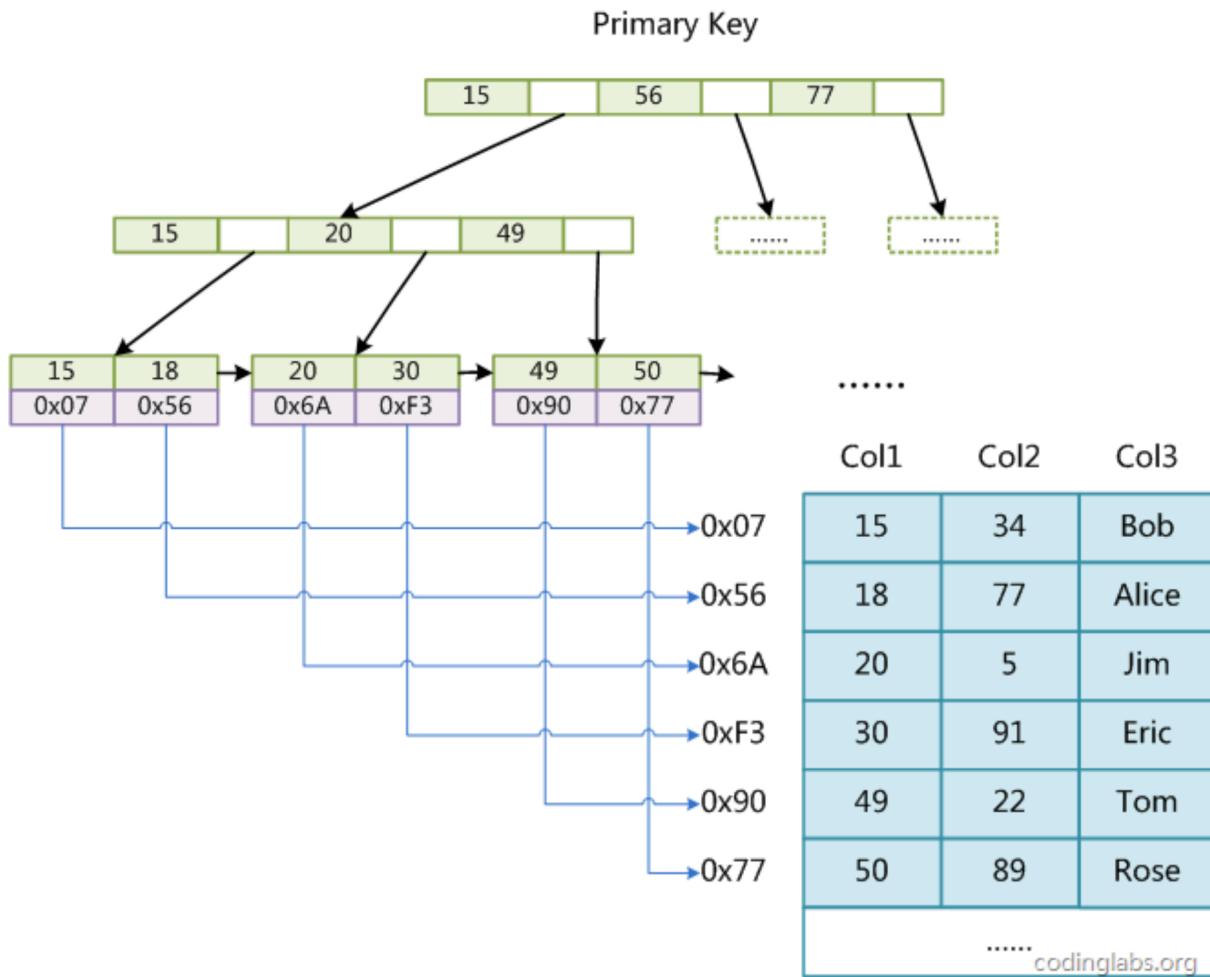
- Hash不支持范围查
- 对于重复键值问题，Hash索引会存在Hash碰撞问题
 - 具体解释：如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应数据；
 - 从示意图中也能看到，如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；
 - 同理，哈希索引也没办法利用索引完成排序，以及like 'xxx%' 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）
 - 哈希索引也不支持多列联合索引的最左匹配规则；B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

2. 不同存储引擎实现

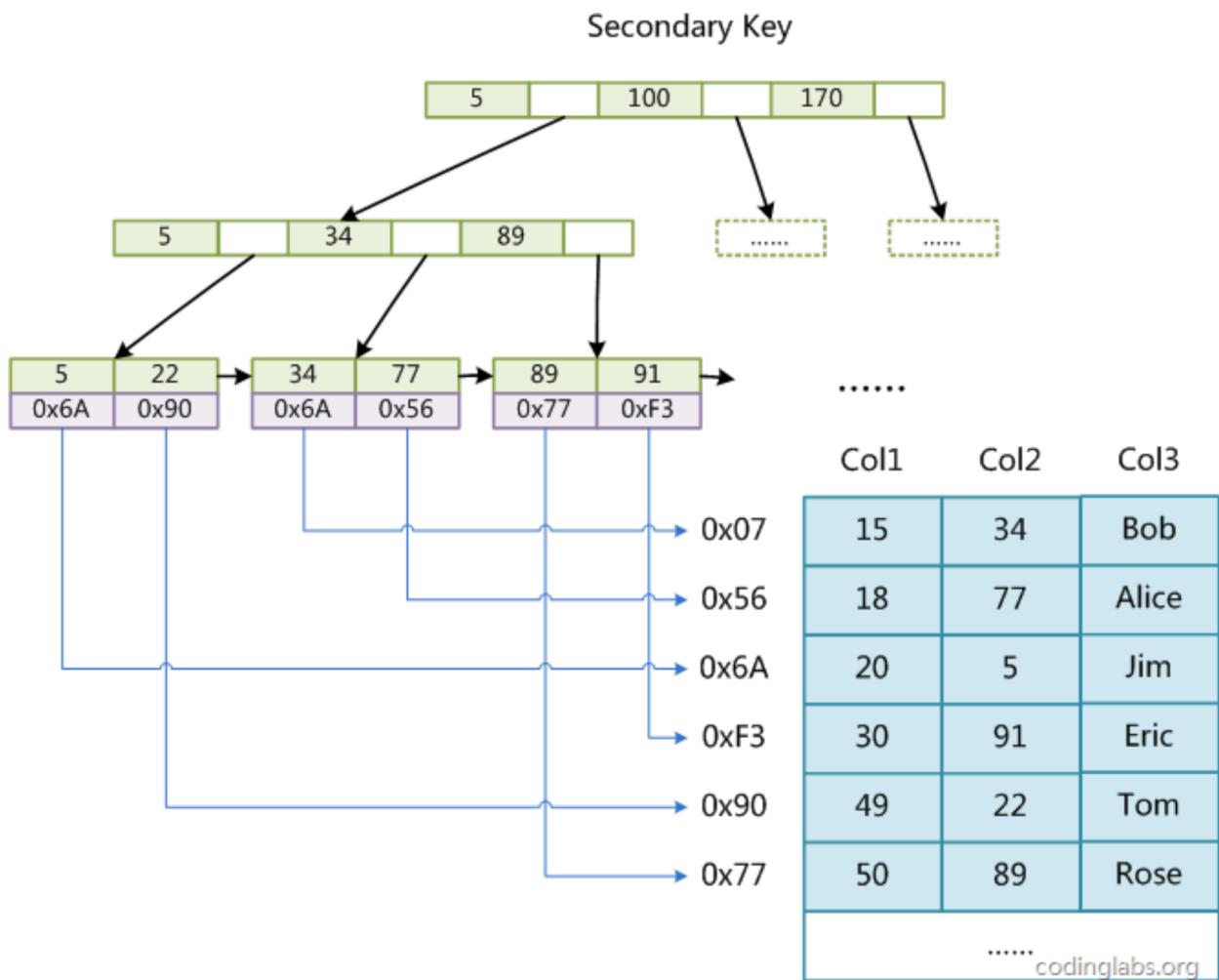
1. MyISAM

给定一张表，其中有一个主键，你能画出在底层具体的存储结构吗？

- 首先要知道，MyISAM索引文件和数据文件是分离的
 - 所以叶子节点存放的是数据文件的相对地址，不是数据文件
 - 从文件也可以看出 .myd 即 my data，表数据文件， .myi 即 my index，索引文件



- 这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

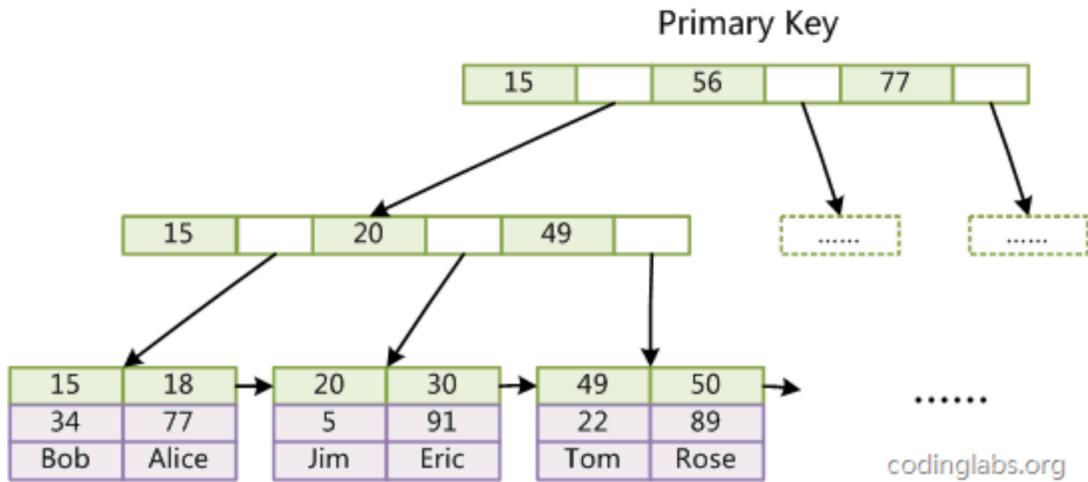
2. InnoDB

给定一张表，如果确认使用的是InnoDB引擎，你能画出底层的数据结构吗？

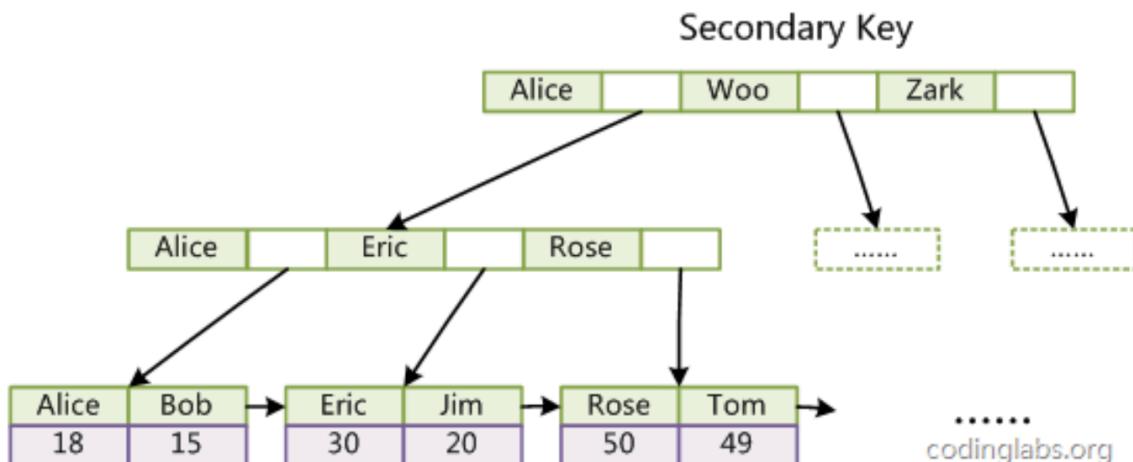
```

CREATE TABLE TEST_INNODB(
    col1 int(11) primary key,
    col2 int(11),
    col3 varchar(20)
)ENGINE=INNODB

```



- 上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整型。
- 第二个与MyISAM的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，上图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

3. 聚集索引和非聚集索引

1. 聚集索引

- 定义
 - 数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。
 - 聚集索引一般是主键索引
 - 聚集索引是一种存储方式，索引的叶子节点就是对应的数据节点（比如对于主键id，id=1所在的叶子节点就存储了id=1,username=小明，score=90的一行数据）

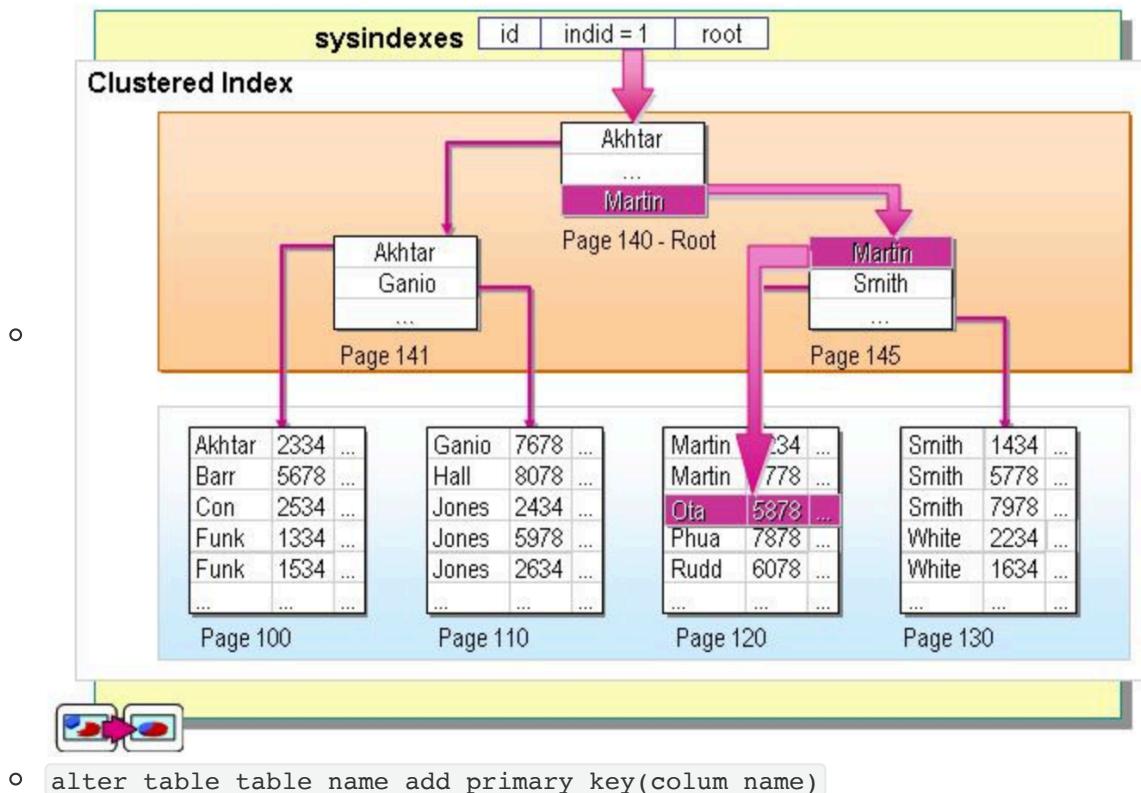
地址	id	username	score
0x01	1	小明	90
0x02	2	小红	80
0x03	3	小华	92
..
0xff	256	小英	70

注：第一列的地址表示该行数据在磁盘中的物理地址，后面三列才是我们SQL里面用的表里的列，其中id是主键，建立了聚集索引。

结合上面的表格就可以理解这句话了吧：数据行的物理顺序与列值的顺序相同，如果我们查询id比较靠后的数据，那么这行数据的地址在磁盘中的物理地址也会比较靠后。而且由于物理排列方式与聚集索引的顺序相同，所以也就只能建立一个聚集索引了。

- 创建聚集索引

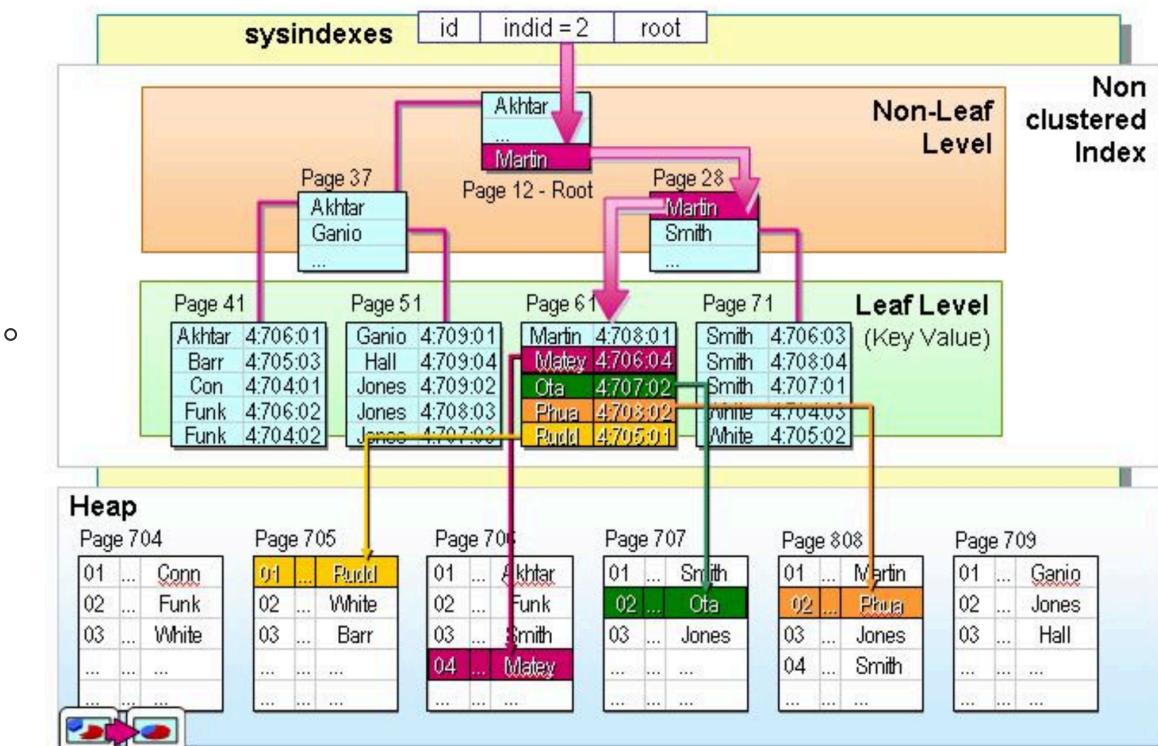
Finding Rows in a Clustered Index



2. 非聚集索引

- 定义
 - 该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

Finding Rows in a Heap with a Nonclustered Index



查询indid = Martin的数据的过程，其中indid为非聚集索引

3. 非聚集索引的二次查询问题

id	username	score
1	小明	90
2	小红	80
3	小华	92
..
256	小英	70

建立聚集索引clustered index(id), 非聚集索引index(username)。

```
# 一次查询
select id, username from t1 where username = '小明'
select username from t1 where username = '小明'
# 需要二次查询score, 消耗更多的时间
select username, score from t1 where username = '小明'
```

参考: <https://www.cnblogs.com/s-b-b/p/8334593.html>

4. 覆盖索引

- 定义

- 覆盖索引（covering index）指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之为实现了索引覆盖。
- 比如建立联合索引index_x(username,score)，上面的二次查询就只需要一次查询就能满足，从索引中直接获取查询结果。

5. 语法层面

1. 创建删除索引

2. 使用Explain语句

```
all < index < range < index_subquery < unique_subquery < index_merge <
ref_or_null < ref < eq_ref < const<system
```

all：这个就是全表扫描了，一般这样的出现这样的SQL而且数据量比较大的话那么就需要进行优化了，要么是这条SQL没有用上索引，要么是没有建立合适的索引。

index：全索引扫描，这个比all效率要好一点，主要有几种情况，一是当前的查询是覆盖索引的，即我们需要的数据在索引中就可以获取（Extra中有Using Index，Extra也是explain的一个字段）（关于覆盖索引：MySQL系列-优化之覆盖索引），或者是使用了索引进行排序，这样就避免数据的重排序（extra中无 Using Index）。如果Extra中Using Index与Using Where同时出现的话，则是利用索引查找键值的意思。

range：这个是index了范围限制，例如>100、<1000之类的查询条件，这样避免的index的全索引扫描，当然限制的范围越小效率就越高。

index_subquery：在某些IN查询中使用此种类型，与unique_subquery类似，但是查询的是非唯一性索引

unique_subquery：在某些IN查询中使用此种类型，而不是常规的ref

index_merge：说明索引合并优化被使用了

ref_or_null：如同ref，但是MySQL必须在初次查找的结果里找出null条目，然后进行二次查找。

ref：使用了非唯一性索引进行数据的查找，例如：我们对用户表的用户名这一列建立了非唯一索引，因为用户名可以重复，当我们查找用户的时候select * from user where username='xxx'的时候就出现了ref，使用非唯一索引查找数据。

eq_ref：这个就很好理解了，使用的唯一性索引进行数据查找，例如主键索引之类的。

const：通常情况下，将一个主键放置到where后面作为条件查询，mysql优化器就能把这次查询优化转化为一个常量，如何转化以及何时转化，这个取决于优化器。这个比eq_ref效率高一点。

system：表只有一行。不过这种情况下就没意义了。

6. 索引分类-谈谈mysql中索引的种类

1. 普通索引

普通索引（由关键字KEY或INDEX定义的索引）的唯一任务是加快对数据的访问速度。

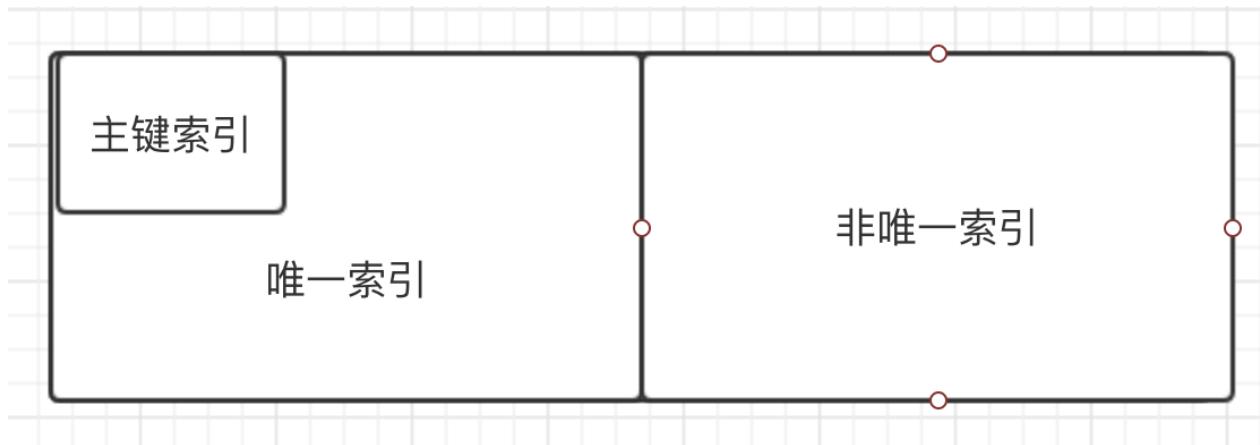
2. 唯一索引

如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。

事实上，在许多场合，人们创建唯一索引的目的往往不是为了提高访问速度，而只是为了避免数据出现重复。

3. 主键索引

主键索引是唯一索引的特殊类型，每个表只能有一个。



1. Hash索引

name	age
Jane	28
Peter	20
David	30

假设使用假想的哈希函数f()，生成对应的设想值： $f('Jane') = 2323$

$f('Peter') = 2456$

$f('David') = 2400$

则哈希索引的数据结构如下：

槽(slot)	值(value)
2323	指向第1行指针
2400	指向第3行指针
2456	指向第2行指针

对于 `select * from user where name = 'Jane'` 那么直接先算 Jane 的哈希值，然后根据 Jane 的 hash 值 2323 去找到对应的第一行数据，查询速度相对于 B-Tree 索引是要快，但是也有一些局限：

- hash索引中只有hash值和行数的指针，因此无法直接使用索引来避免读取行，但是因为这种索引读取快，性能影响不明显。

- hash索引不是按照索引值顺序存储，无法使用于排序。
- 不支持部分列匹配查找，这里面是使用索引列的全部内容来计算哈希值，例如(A,B)两列一起建索引，单纯使用A一列，那么就无法使用索引，B-Tree索引的话，因为支持匹配最左前缀，所以这种情况适用性偏好。
- 哈希索引只支持等值查询，包括=、in()、<=>，不支持where age > 10 这种范围查询。
- 哈希冲突很多的话，维护索引操作的代价也很高

7. 最左匹配原则

1. explain

- explain的type类型
 - index：这种类型表示是mysql会对整个该索引进行扫描。要想用到这种类型的索引，对这个索引并无特别要求，只要是索引，或者某个复合索引的一部分，mysql都可能会采用index类型的方式扫描。但是呢，缺点是效率不高，mysql会从索引中的第一个数据一个个的查找到最后一个数据，直到找到符合判断条件的某个索引。
 - ref：这种类型表示mysql会根据特定的算法快速查找到某个符合条件的索引，而不是会对索引中每一个数据都进行一一的扫描判断，也就是所谓你平常理解的使用索引查询会更快的取出数据。而要想实现这种查找，索引却是有要求的，要实现这种能快速查找的算法，索引就要满足特定的数据结构。简单说，也就是索引字段的数据必须是有序的，才能实现这种类型的查找，才能利用到索引。

2. 使用案例

```
CREATE TABLE `user2` (
  `userid` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(20) NOT NULL DEFAULT '',
  `password` varchar(20) NOT NULL DEFAULT '',
  `usertype` varchar(20) NOT NULL DEFAULT '',
  PRIMARY KEY (`userid`),
  KEY `a_b_c_index` (`username`, `password`, `usertype`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

`a_b_c_index` 实际建立了(username)、(username,password)、(username、password、usertype) 三个索引

```
//不使用索引
explain select * from user2 where password = '1';
//使用索引
explain select * from user2 where username = '1' and password = '1';
//使用索引-即使是乱序
explain select * from user2 where password = '1' and username = '1';
```

3. 失效的情况

- 最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如`a = 1 and b = 2 and c > 3 and d = 4`如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
- =和in可以乱序，比如`a = 1 and b = 2 and c = 3`建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

4. 为什么要使用联合索引

减少开销。建一个联合索引(col1,col2,col3)，实际相当于建了(col1),(col1,col2),(col1,col2,col3)三个索引。每多一个索引，都会增加写操作的开销和磁盘空间的开销。对于大量数据的表，使用联合索引会大大的减少开销！

覆盖索引。对联合索引(col1,col2,col3)，如果有如下的sql: `select col1,col2,col3 from test where col1=1 and col2=2`。那么MySQL可以直接通过遍历索引取得数据，而无需回表，这减少了很多的随机io操作。减少io操作，特别的随机io其实是dba主要的优化策略。所以，在真正的实际应用中，覆盖索引是主要的提升性能的优化手段之一。

效率高。索引列越多，通过索引筛选出的数据越少。有1000W条数据的表，有如下sql:`select from table where col1=1 and col2=2 and col3=3`,假设每个条件可以筛选出10%的数据，如果只有单值索引，那么通过该索引能筛选出1000W10%=100w条数据，然后再回表从100w条数据中找到符合col2=2 and col3= 3的数据，然后再排序，再分页；如果是联合索引，通过索引筛选出1000w10% 10% *10%=1w，效率提升可想而知！

8. 索引失效的情况

2. 事务★★★★★

事务是对数据库中一系列操作进行统一的回滚或者提交的操作，主要用来保证数据的完整性和一致性。

1. 事务的特性【ACID】是什么

- 原子性 (Atomicity) :原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须完全应用到数据库，如果操作失败则不能对数据库有任何影响。
- 一致性 (Consistency) :事务开始前和结束后，数据库的完整性约束没有被破坏。比如A向B转账，不可能A扣了钱，B却没收到。
- 隔离性 (Isolation) :隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
- 持久性 (Durability) :持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

2. 事务并发操作会带来的问题

1. 脏读

事务A：更新了数据行X为Y，并没有提交

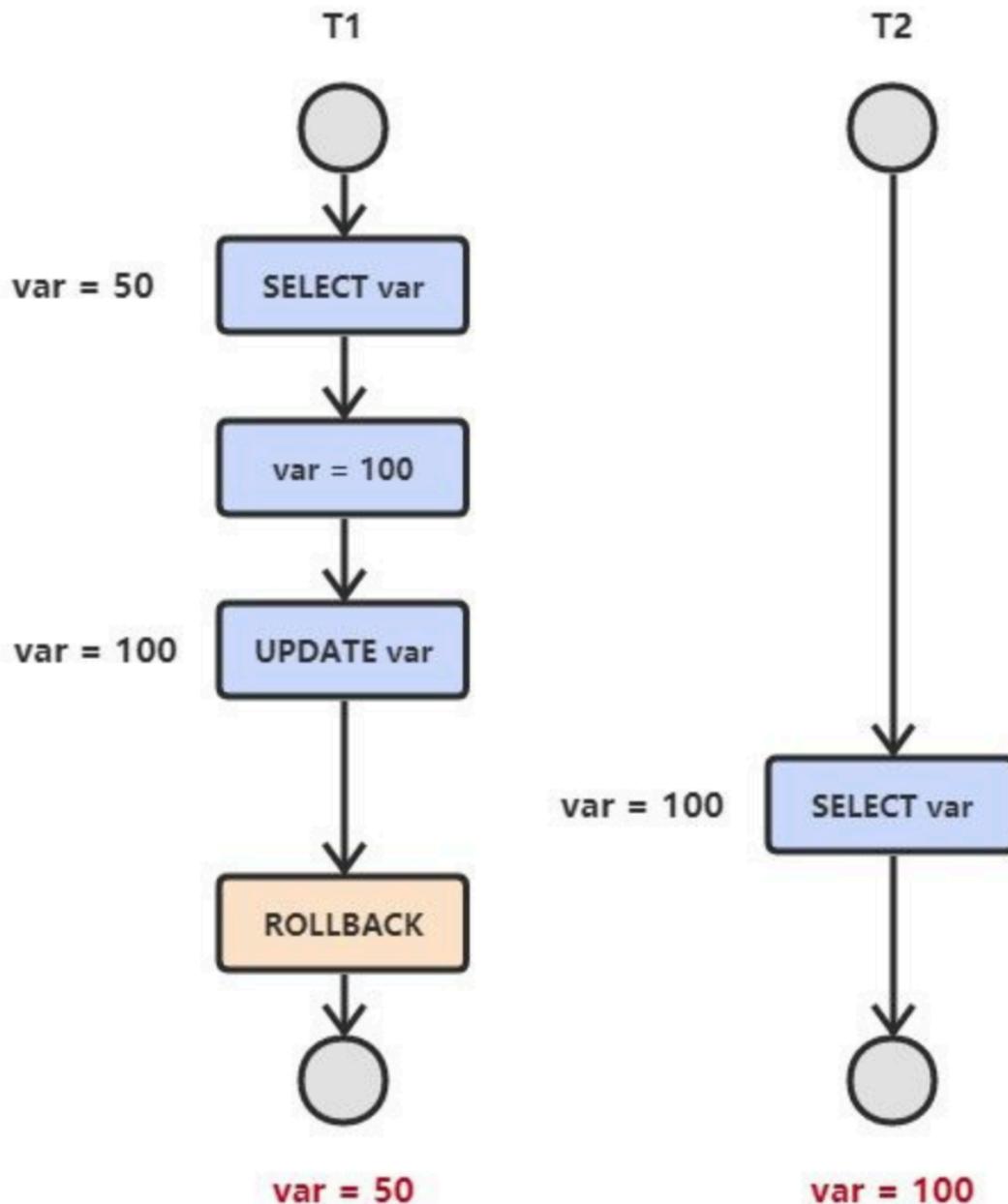
事务B：读取数据行X，观察为Y，发现数据改变

事务A：回滚操作

事务B：读取数据行X，观察为X，那么Y就是脏数据

脏读又称无效数据读出。一个事务读取另外一个事务还没有提交的数据叫脏读。

(贪心， 读到了没有提交的数据)



2. 不可重复读

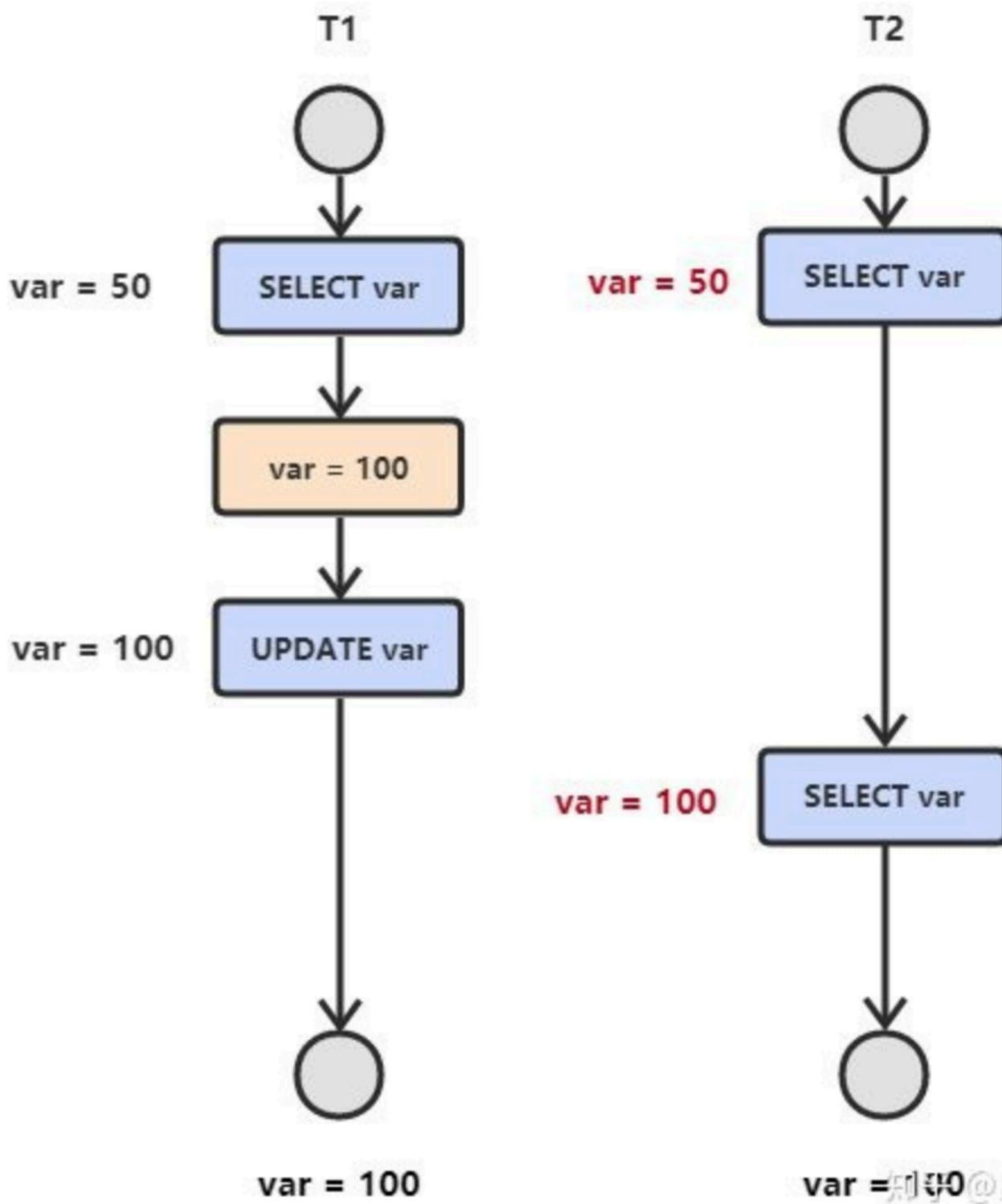
事务A：读取数据行X

事务B：修改数据行X为Y，提交事务

事务A：读取数据行X，观察为Y，那么在同一个事务A，读到了两个不同的数据。

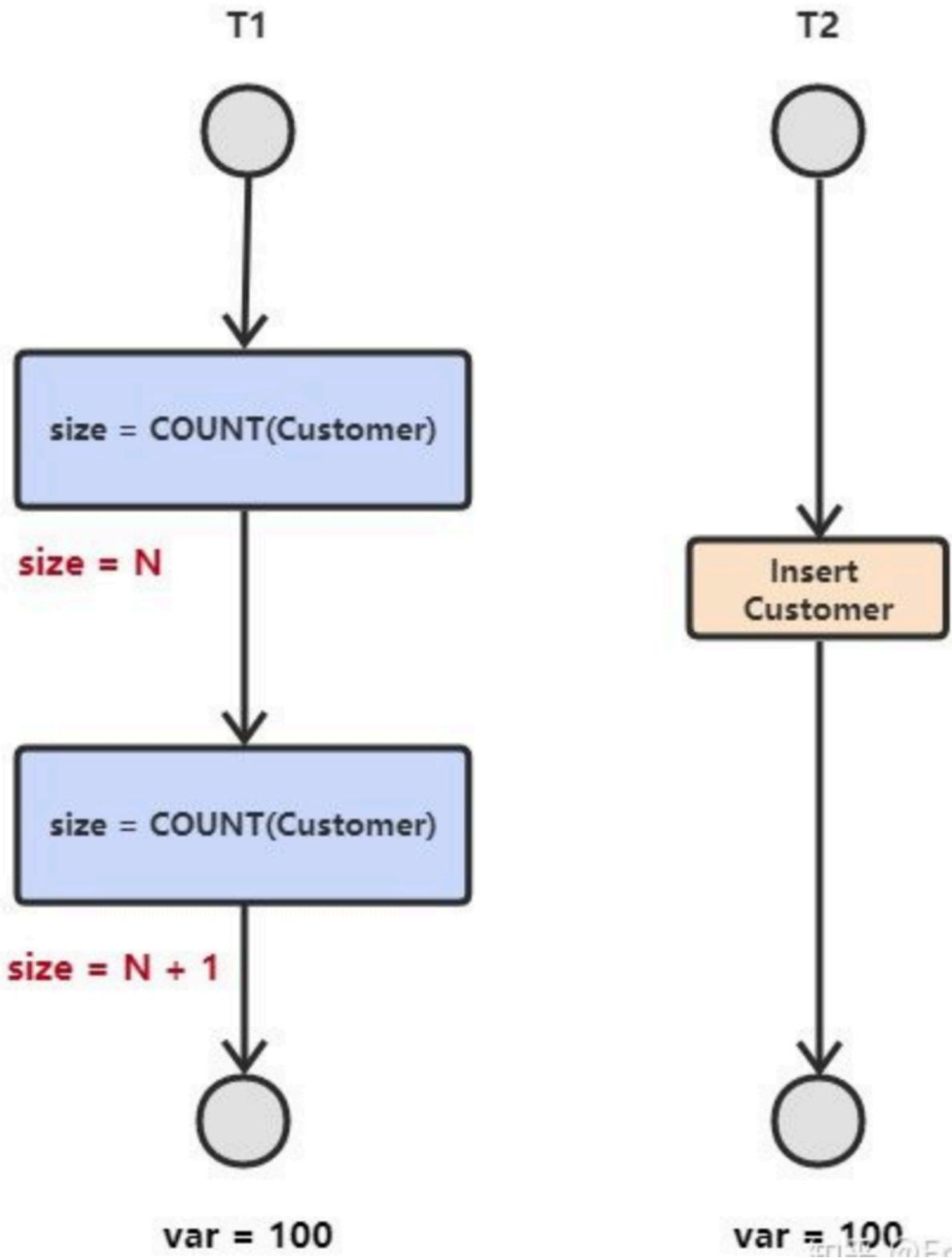
在一个事务中前后两次读取的结果并不致，导致了不可重复读。

(事务读取之间，有其他事务提交操作，所以不能重复读到相同的数据)



3. 幻读

事务 T1 读取一条指定的 Where 子句所返回的结果集，然后 T2 事务新插入一行记录，这行记录恰好可以满足 T1 所使用的查询条件。然后 T1 再次对表进行检索，但又看到了 T2 插入的数据。（和可重复读类似，但是事务 T2 的数据操作仅仅是插入和删除，不是修改数据，读取的记录数量前后不一致）



4. 幻读和不可重复读有什么区别

幻读主要是针对插入和删除操作，不可重复读主要是针对修改操作。

5. 脏读和不可重复读有什么区别

脏读主要是针对没有提交的事务的修改，不可重复读主要是针对事务提交之后的修改。

3. 事务的隔离级别有哪些

首先是了解了数据库在并发操作下会引发的问题，然后再来了解隔离级别，针对哪些问题

1. 读未提交 (Read Uncommitted)

最低的隔离等级，允许其他事务看到没有提交的数据，会导致脏读。

2. 读已提交 (Read Committed)

被读取的数据可以被其他事务修改，这样可能导致不可重复读。也就是说，事务读取的时候获取读锁，但是在读完之后立即释放(不需要等事务结束)，而写锁则是事务提交之后才释放，释放读锁之后，就可能被其他事务修改数据。该等级也是 SQL Server 默认的隔离等级。

3. 可重复读 (Repeatable Read)

MySQL的默认事务隔离级别

所有被 **Select** 获取的数据都不能被修改，这样就可以避免一个事务前后读取不一致的情况。但是没有办法控制幻读，因为这个时候其他事务不能更改所选的数据，但是可以增加数据，因为强恶意事务没有范围锁。

4. 串行化 (Serializable)

所有事务一个接着一个的执行，这样可以避免幻读 (**phantom read**)，对于基于锁来实现并发控制的数据库来说，串行化要求在执行范围查询的时候，需要获取范围锁，如果不是基于锁实现并发控制的数据库，则检查到有违反串行操作的事务时，需回滚该事务。

5. 四种隔离级别的区别

- (1) 设置read uncommitted的时候可以发现脏读[事务A读到了事务B没有提交的数据]
- (2) 设置read committed的时候可以解决上述问题，但是在A中同一个事务，读到了不同的数据，导致了[不可重复读]
- (3) 设置repeatable read的时候，如果在A中插入一条数据，提交事务之后，在B中无法观察到该数据，所以产生了[幻读]
- (4) 设置Serializable的时候，是最安全但是效率最低的。

6. 可重复读-实现原理MVCC

MySQL通过MVCC(multi version concurrent control)来实现默认的"Repeatable Read"事务隔离级别

1. MVCC定义

MVCC，**Multi-Version Concurrency Control**，多版本并发控制。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。(乐观锁实现的一种机制)如果有人从数据库中读数据的同时，有另外的人写入数据，有可能读数据的人会看到『半写』或者不一致的数据。有很多种方法来解决这个问题，叫做并发控制方法。最简单的方法，通过加锁，让所有的读者等待写者工作完成，但是这样效率会很差。MVCC 使用了一种不同的手段，每个连接到数据库的读者，在某个瞬间看到的是数据库的一个快照，写者写操作造成的变化在写操作完成之前（或者数据库事务提交之前）对于其他的读者来说是不可见的。当一个 MVCC 数据库需要更一个一条数据记录的时候，它不会直接用新数据覆盖旧数据，而是将旧数据标记为过时 (**obsolete**) 并在别处增加新版本的数据。这样就会有存储多个版本的数据，但是只有一个是最新的。这种方式允许读者读取在他读之前已经存在的数据，即使这些在读的过程中半路被别人修改、删除了，也对先前正在读的用户没有影响。这种多版本的方式避免了填充删除操作在内存和磁盘存储结构造成的空洞的开销，但是需要系统周期性整理 (sweep through) 以真实删除老的、过时的数据。对于面向文档的数据库 (Document-oriented)

database, 也即半结构化数据库) 来说, 这种方式允许系统将整个文档写到磁盘的一块连续区域上, 当需要更新的时候, 直接重写一个版本, 而不是对文档的某些比特位、分片切除, 或者维护一个链式的、非连续的数据库结构。

2. mysql底层实现

InnoDB在每行记录后面保存两个隐藏的列来, 分别保存了这个行的创建时间和行的删除时间。这里存储的并不是实际的时间值, 而是系统版本号, 当数据被修改时, 版本号加1。在读取事务开始时, 系统会给当前读事务一个版本号, 事务会读取版本号<=当前版本号的数据。此时如果其他写事务修改了这条数据, 那么这条数据的版本号就会加1, 从而比当前读事务的版本号高, 读事务自然而然的就读不到更新后的数据了。

举个栗子, 假设初始版本号为1:

INSERT

```
insert into user (id,name) values (1,'Tom');
```

id	name	create_version	delete_version
1	Tom	1	

下面模拟一下文章开头的场景:

SELECT (事务A)

```
select * from user where id = 1;
```

此时读到的版本号为1, 值为"Tom"

UPDATE(事务B)

```
update user set name = 'Jerry' where id = 1;
```

在更新操作的时候, 该事务的版本号在原来的基础上加1, 所以版本号为2。先将要更新的这条数据标记为已删除, 并且删除的版本号是当前事务的版本号, 然后插入一行新的记录

id	name	create_version	delete_version
1	Tom	1	2
1	Jerry	2	

SELECT (事务A)

此时事务A再重新读数据:

```
select * from user where id = 1;
```

由于事务A一直没提交，所以此时读到的版本号还是为1，所以读到的还是Tom这条数据，也就是可重复读。

3. MySQL中的锁

- 为什么要加锁
 - 多线程环境的统一解决方案
 - 当多个用户并发地存取数据时，在数据库中就可能会产生多个事务同时操作同一行数据的情况，若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据的一致性。

(1) 服务级别—表锁

实际上就是读写锁

表锁可以是显式也可以是隐式的。显示的锁用Lock Table来创建，但要记得Lock Table之后进行操作，需要在操作结束后，使用UnLock来释放锁。Lock Tables有read和write两种，Lock Tables.....Read通常被称为共享锁或者读锁，读锁或者共享锁，是互相不阻塞的，多个用户可以同一时间使用共享锁互相不阻塞。Lock Table.....write通常被称为排他锁或者写锁，写锁或者排他锁会阻塞其他的读锁或者写锁，确保在给定时间里，只有一个用户执行写入，防止其他用户读取正在写入的同一资源。

1. Lock Tables....READ不会阻塞其他线程对表数据的读取，会阻塞其他线程对数据变更
2. Lock Tables....WRITE会阻塞其他线程对数据读和写
3. Lock Tables....READ不允许对表进行更新操作(新增、删除也不行)，**并且不允许访问未被锁住的表**
4. Lock Tables....WRITE允许对被锁住的表进行增删改查，但不允许对其他表进行访问

<https://www.cnblogs.com/null-qige/p/8664009.html>

(2) InnoDB锁

锁优化：<https://www.zybuluo.com/mikumikulch/note/783493>

```
CREATE TABLE `test_index_table` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) DEFAULT NULL,
  `birthday` datetime DEFAULT NULL,
  `address` varchar(45) DEFAULT NULL,
  `phone` varchar(45) DEFAULT NULL,
  `note` varchar(45) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `NAME_ADDRESS` (`name`, `address`) USING BTREE,
  KEY `AGE` (`age`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=283 DEFAULT CHARSET=utf8;

insert into test_index_table (id, name, birthday, address, phone, note, age) values
(100, '王二', now(), '成都', 18011193339, '备注', 26);
```

```
insert into test_index_table (id,name,birthday,address,phone,note,age) values  
(200,'王二',now(),'北京',18011193339,'备注',26);  
  
select * from test_index_table where age = 26 lock in share mode;  
select * from test_index_table where age = 26 for update;
```

1. 行锁

1. 引擎：行锁存在于InnoDB引擎，MyISAM引擎只有表锁
2. 优点：锁定行，处理并发的能力更强

```
// 演示行锁  
drop table if exists test_innodb_lock;  
CREATE TABLE test_innodb_lock (  
    a INT (11),  
    b VARCHAR (20)  
) ENGINE INNODB DEFAULT charset = utf8;  
insert into test_innodb_lock values (1,'a');  
insert into test_innodb_lock values (2,'b');  
insert into test_innodb_lock values (3,'c');  
insert into test_innodb_lock values (4,'d');  
insert into test_innodb_lock values (5,'e');  
  
create index idx_lock_a on test_innodb_lock(a);  
create index idx_lock_b on test_innodb_lock(b);  
  
// 关闭数据库的自动提交  
set autocommit=0;
```

①在A会话中执行如下语句。

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test_innodb_lock where a=7 for update;
+-----+-----+
| a    | b    |
+-----+-----+
| 7   | 6000 |
+-----+-----+
1 row in set (0.00 sec)
```

此时就锁定了a=7这一行。

②在B会话中对该行进行更新操作。

```
mysql> update test_innodb_lock set b='xxx' where a=7;
| ←                                         B会话阻塞
```

只有在A会话中进行了commit后，B会话的更新操作才能执行。

```
mysql> update test_innodb_lock set b='xxx' where a=7;
Query OK, 1 row affected (3.55 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

①InnoDB存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会更高一些（多个锁，一个锁），但是在整体并发处理能力方面要远远优于MyISAM的表级锁定。当系统处于高并发量的时候，InnoDB的整体性能和MyISAM相比就会有比较明显的优势了。

②InnoDB的行锁定同样尤其脆弱的一面（间隙锁危害），当使用不当时可能会让InnoDB的整体性能表现不仅不能比MyISAM高，甚至可能更差。

2. 间隙锁【行锁的问题】

- 定义

- 当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁，对于键值在条件范围内但不存在的记录，叫作“间隙（GAP）”。
- InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁。（Next-Key锁）
- 因为在Query执行过程中通过范围查找的话，会锁定整个范围内的所有索引键值，即使这个索引不存在。间隙锁有一个比较致命的弱点，就是当锁定一个范围键值后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定值范围内的任何数据。在某些场景下这个可能会对性能造成很大的危害。
- 就是会锁定一个范围。

- 例子

- 首先关闭MySQL的自动提交
- 然后在A会话中更新了某列的数据(必须要使用索引)比如从a到b
- 在B会话中插入该列的数据c(a < c < b)，会造成阻塞（也就是另外一个会话加锁的提现）

```
// 演示间隙锁
drop table if exists test_innodb_lock;
CREATE TABLE test_innodb_lock (
    a INT (11),
    b VARCHAR (20)
```

```

) ENGINE INNODB DEFAULT charset = utf8;
insert into test_innodb_lock values (1,1000);
insert into test_innodb_lock values (3,3000);
insert into test_innodb_lock values (4,4000);
insert into test_innodb_lock values (5,5000);
insert into test_innodb_lock values (7,7000);
insert into test_innodb_lock values (9,9000);

create index idx_lock_a on test_innodb_lock(a);
create index idx_lock_b on test_innodb_lock(b);

```

3. 死锁

4. 存储引擎

1. MyISAM与InnoDB的区别，以及使用场景

	MyISAM	InnoDB
构成上的区别:	每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。 .frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。	基于磁盘的资源是InnoDB表空间数据文件和它的日志文件，InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB
事务处理上方面	MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快，但是不提供事务支持	InnoDB提供事务支持事务，外部键 (foreign key) 等高级数据库功能
SELECT UPDATE,INSERT, Delete操作	如果执行大量的SELECT， MyISAM是更好的选择	1.如果你的数据执行大量的 INSERT** 或 UPDATE ，出于性能方面的考虑，应该使用 InnoDB表 2. DELETE FROM table 时， InnoDB 不会重新建立表，而是一行一行的删除。 3. LOAD TABLE FROM MASTER** 操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性

		(例如外键) 的表不适用
对 AUTO_INCREMENT 的操作	<p>每表一个AUTO_INCREMENT列的内部处理。 MyISAM** 为INSERT和UPDATE操作自动更新这一列**。这使得AUTO_INCREMENT列更快(至少10%)。在序列顶的值被删除之后就不能再利用。(当AUTO_INCREMENT列被定义为多列索引的最后一列, 可以出现重使用从序列顶部删除的值的情况)。AUTO_INCREMENT值可用ALTER TABLE或myisamch来重置。对于AUTO_INCREMENT类型的字段, InnoDB中必须包含只有该字段的索引, 但是在MyISAM表中, 可以和其他字段一起建立联合索引更好和更快的auto_increment处理</p>	<p>如果你为一个表指定 AUTO_INCREMENT 列, 在数据词典里的 InnoDB 表句柄包含一个名为自动增长计数器的计数器, 它被用在为该列赋新值。自动增长计数器仅被存储在主内存中, 而不是存在磁盘上关于该计算器的算法实现, 请参考 AUTO_INCREMENT** 列在InnoDB里如何工作**</p>
表的具体行数	<p><code>select count() from table, MyISAM</code> 只要简单的读出保存好的行数, 注意的是, 当 <code>count()</code> 语句包含 <code>where</code> 条件时, 两种表的操作是一样的</p>	InnoDB 中不保存表的具体行数, 也就是说, 执行 <code>select count(*) from table</code> 时, InnoDB 要扫描一遍整个表来计算有多少行
锁	表锁	<p>提供行锁(locking on row level), 提供与 Oracle 类型一致的不加锁读取(non-locking read in SELECTs), 另外, InnoDB 表的行锁也不是绝对的, 如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围, InnoDB 表同样会锁全表, 例如 <code>update table set num=1 where name like "%aaa%"</code></p>

- 一、 **InnoDB 支持事务, MyISAM 不支持**, 这一点是非常之重要。事务是一种高级的处理方式, 如在一些列增删改中只要哪个出错还可以回滚还原, 而 MyISAM 就不可以了。
- 二、 MyISAM 适合查询以及插入为主的应用, InnoDB 适合频繁修改以及涉及到安全性较高的应用
- 三、 **InnoDB 支持外键, MyISAM 不支持**

四、MyISAM是默认引擎，InnoDB需要指定

五、InnoDB不支持FULLTEXT类型的索引

六、InnoDB中不保存表的行数，如select count() from table时，InnoDB需要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count()语句包含where条件时MyISAM也需要扫描整个表

七、对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引

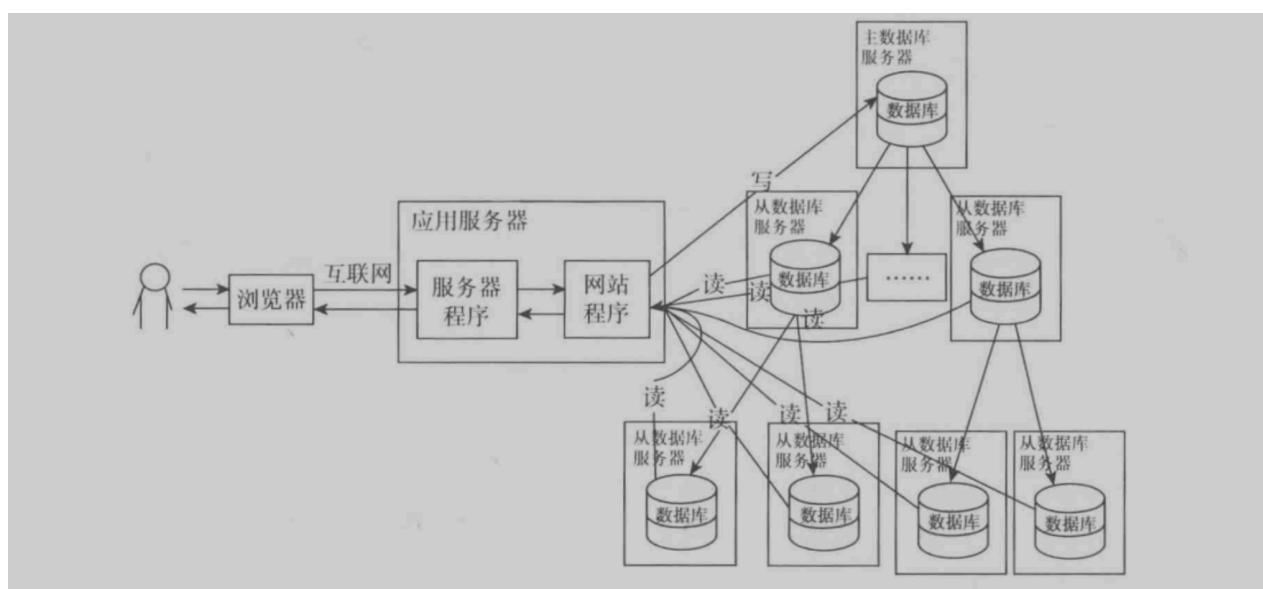
八、清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表

九、InnoDB支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'

5. 读写分离

1. 什么是『读写分离』

先上一张图，这个图是书里面的，说起来很简单就是把读取数据和增删改数据分离到不同数据库里面。增删改放到主数据库里面，读取数据则是放到从数据库里面。主数据的数据通过底层同步到从数据库里面。



大多数互联网业务，往往读多写少，这时候，数据库的读会首先称为数据库的瓶颈，这时，如果我们希望能够线性的提升数据库的读性能，消除读写锁冲突从而提升数据库的写性能，那么就可以使用“分组架构”（读写分离架构）。

用一句话概括，读写分离是用来解决数据库的读性能瓶颈的。

2. 读写分离实现的具体操作

7. MySQL中的数据结构

整形：

整数类型	字节	最小值	最大值
TINYINT	1	有符号-128 无符号 0	有符号 127 无符号 255
SMALLINT	2	有符号-32768 无符号 0	有符号 32767 无符号 65535
MEDIUMINT	3	有符号-8388608 无符号 0	有符号 8388607 无符号 1677215
INT、INTEGER	4	有符号-2147483648 无符号 0	有符号 2147483647 无符号 4294967295
BIGINT	8	有符号-9223372036854775808 无符号 0	有符号 9223372036854775807 无符号 18446744073709551615

浮点型：

MySQL数据类型	含义
float(m,d)	单精度浮点型 8位精度(4字节) m总个数, d小数位
double(m,d)	双精度浮点型 16位精度(8字节) m总个数, d小数位

字符类型：

MySQL数据类型	含义
char(n)	固定长度, 最多 $2^8 - 1$ 个字符, $2^8 - 1$ 个字节
varchar(n)	可变长度, 最多 $2^{16} - 1$ 个字符, $2^{16} - 1$ 个字节

日期时间类型：

MySQL数据类型	含义
date	日期 '2008-12-2'
time	时间 '12:25:36'
datetime	日期时间 '2008-12-2 22:06:44'
timestamp	自动存储记录修改时间

1. int(11)的含义

```

CREATE TABLE `test_int` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `i1` int(11) unsigned zerofill DEFAULT NULL,
    `i2` int(3) unsigned zerofill DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

insert into test_int (i1,i2) values (1,1);
insert into test_int (i1,i2) values (123456,123456);
insert into test_int (i1,i2) values (1234567890,1234567890);

```

```

mysql> select * from test_int;
+---+-----+-----+
| id | i1   | i2   |
+---+-----+-----+
| 1  | 0000000001 | 001  |
| 2  | 00000123456 | 123456 |
| 3  | 01234567890 | 1234567890|
+---+-----+-----+

```

- int(11)和int(3)的区别在于
 - 当小于11位，3位时会补零显示
 - 不影响实际存储的精度

2. double(m,d)的含义

```

create table decimal_test(
    id int auto_increment PRIMARY key,
    score decimal(5,2) -- 取值范围是 -999.99 到 999.99
);

insert into decimal_test(score) VALUES(1.23); -- 1.23
insert into decimal_test(score) VALUES(123.45); -- 123.45
insert into decimal_test(score) VALUES(123.455); -- 123.46
insert into decimal_test(score) VALUES(123.451); -- 123.45
insert into decimal_test(score) VALUES(123.451123); -- 123.45
insert into decimal_test(score) VALUES(12345.451123); -- Out of range

```

- m-d表示整数的限制
- d表示小数的最高显示精度

3. char和varchar的区别

下面的表显示了将各种字符串值保存到CHAR(4)和VARCHAR(4)列后的结果，说明了CHAR和VARCHAR之间的差别：

值	CHAR(4)	存储需求	VARCHAR(4)	存储需求
''	' '	4个字节	''	1个字节
'ab'	'ab '	4个字节	'ab '	3个字节
'abcd'	'abcd'	4个字节	'abcd'	5个字节
'abcdefgh'	'abcd'	4个字节	'abcd'	5个字节

4. timestamp用法

```
CREATE TABLE categories (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO categories(name) VALUES ('A');
```

```
+----+----+-----+
| id | name | created_at      |
+----+----+-----+
| 1  | A    | 2019-07-23 01:41:19 |
+----+----+-----+
```

左连接和右连接

```
DROP TABLE `a_table`;
DROP TABLE `b_table`;

CREATE TABLE `a_table` (
    `a_id` int(11) PRIMARY KEY AUTO_INCREMENT,
    `a_name` varchar(10) DEFAULT NULL,
    `a_part` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `b_table` (
    `b_id` int(11) PRIMARY KEY AUTO_INCREMENT,
    `b_name` varchar(10) DEFAULT NULL,
    `b_part` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into a_table (a_name,a_part) values ('a1','ap1');
insert into a_table (a_name,a_part) values ('a2','ap2');
insert into a_table (a_name,a_part) values ('a3','ap3');

insert into b_table (b_name,b_part) values ('a1','ap1');
```

```
insert into b_table (b_name,b_part) values ('a2','ap2');
insert into b_table (b_name,b_part) values ('b3','bp3');
```

- 内连接
 - `select * from A INNER JOIN B ON 条件`
 - 返回两个集合交集的部分

```
mysql> select * from a_table inner join b_table on a_table.a_name = b_table.b_name;
+----+-----+-----+----+-----+
| a_id | a_name | a_part | b_id | b_name | b_part |
+----+-----+-----+----+-----+
| 1   | a1    | ap1   | 1   | a1    | ap1   |
| 2   | a2    | ap2   | 2   | a2    | ap2   |
+----+-----+-----+----+-----+
2 rows in set (0.00 sec)
```

- 左连接
 - `select * from A left join B on 条件`
 - 左表会显示全部元素，右表会显示符合条件的部分+其他不符合条件的部分[NULL表示]

```
mysql> select * from a_table a left join b_table b on a.a_name = b.b_name;
+----+-----+-----+----+-----+
| a_id | a_name | a_part | b_id | b_name | b_part |
+----+-----+-----+----+-----+
| 1   | a1    | ap1   | 1   | a1    | ap1   |
| 2   | a2    | ap2   | 2   | a2    | ap2   |
| 3   | a3    | ap3   | NULL | NULL  | NULL   |
+----+-----+-----+----+-----+
3 rows in set (0.00 sec)
```

- 右连接
 - `select * from A right join B on 条件`
 - 右表会显示全部，左表会显示符合提交的部分+其他不符合条件的部分[NULL表示]

```
mysql> select * from a_table a right join b_table b on a.a_name = b.b_name;
+----+-----+-----+----+-----+
| a_id | a_name | a_part | b_id | b_name | b_part |
+----+-----+-----+----+-----+
| 1   | a1    | ap1   | 1   | a1    | ap1   |
| 2   | a2    | ap2   | 2   | a2    | ap2   |
| NULL | NULL  | NULL  | 3   | b3    | bp3   |
+----+-----+-----+----+-----+
3 rows in set (0.00 sec)
```

8. 主从同步以及原理

9. 数据库范式

1. 第一范式-【原子性】

- 第一范式 (1NF) 要求数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。

- 若某一列有多个值，可以将该列单独拆分成一个实体，新实体和原实体间是一对多的关系。
- 在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

2. 第二范式-一种表只能保存一种数据

- 满足第二范式（2NF）必须先满足第一范式（1NF）。
- 第二范式要求实体中每一行的所有非主属性都必须完全依赖于主键；即：**非主属性必须完全依赖于主键。**
- 完全依赖：主键可能由多个属性构成，完全依赖要求不允许存在非主属性依赖于主键中的某一部分属性。
- 若存在哪个非主属性依赖于主键中的一部分属性，那么要将发生部分依赖的这一组属性单独新建一个实体，并且在旧实体中用外键与新实体关联，并且新实体与旧实体间是一对多的关系。

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

3. 第三范式-主键相关

- 满足第三范式必须先满足第二范式。
- 第三范式要求：实体中的属性不能是其他实体中的非主属性。因为这样会出现冗余。即：属性不依赖于其他非主属性。
- 如果一个实体中出现其他实体的非主属性，可以将这两个实体用外键关联，而不是将另一张表的非主属性直接写在当前表中。

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

10. UUID可以作为主键吗

应用问题

DataGrip-Mac

1. distinct关键词

这是一张成绩单表，通过一条sql查询出所有学科都及格（60分）的学生。

我写了的用到了子查询，然后问我不用子查询怎么做？

姓名	学科	分数
张三	英语	60
张三	数学	70
张三	语文	58
李四	英语	80

```
//创建测试用例
create table Student(
    id int(11) primary key auto_increment,
```

```

name varchar(20),
course varchar(20),
score int(11)
)ENGINE=InnoDB DEFAULT CHARSET=utf8 ;

insert into Student (name,course,score) values ('张三','数学',60);
insert into Student (name,course,score) values ('张三','英语',70);
insert into Student (name,course,score) values ('张三','语文',50);
insert into Student (name,course,score) values ('李四','英语',80);

select distinct name from Student where score > 60;

```

2. 第二高薪水

Write a SQL query to get the second highest salary from the Employee table.

Id	Salary
1	100
2	200
3	300

```

# 从所有小于最大值的salary中找到的最大的—second
SELECT MAX(Salary) FROM Employee
Where Salary <
(SELECT MAX(Salary) FROM Employee);

```

3. 学生平均分

group by

我们需要得到所有功课平均分达到60分的同学和他们的均分：

[<https://www.cnblogs.com/hhe0/p/9556070.html>]

```

CREATE TABLE `courses` (
`id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '自增id',
`student` VARCHAR(255) DEFAULT NULL COMMENT '学生',
`class` VARCHAR(255) DEFAULT NULL COMMENT '课程',
`score` INT(255) DEFAULT NULL COMMENT '分数',
PRIMARY KEY (`id`),
UNIQUE KEY `course` (`student`, `class`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

SELECT `student`, AVG(`score`) AS `avg_score`

```

```
FROM `courses`  
GROUP BY `student`  
HAVING AVG(`score`) >= 60  
ORDER BY `avg_score` DESC;
```

0x7 Web开发

001 基础知识

1. Cookie和Session

1. Cookie和Session的区别

0. Cookie和Session的出现是为了解决什么问题—HTTP协议
1. Cookie和Session都是一种会话机制
2. 保存的位置不同: Cookie保存在客户端, Session保存在服务器
3. Java中的Cookie
 - 3.1 `Cookie cookie = new Cookie(String name, String value) //设置Cookie`
4. Java中的Session
 - 4.1 `HttpSession session = httpServletRequest.getSession() //创建|获取Session`
 - 4.2 `String username = (String) session.getAttribute(String name) //获取Session某个属性`
5. Cookie保存在本地, 相对不安全, Session相对安全

1. 存储位置不同

cookie的数据信息存放在客户端浏览器上, session的数据信息存放在服务器上。

2. 存储容量不同

单个cookie保存的数据<=4KB, 一个站点最多保存20个Cookie。

对于session来说并没有上限, 但出于对服务器端的性能考虑, session内不要存放过多的东西, 并且设置session删除机制。

3. 安全性不同

cookie对客户端是可见的, 别有用心的人可以分析存放在本地的cookie并进行cookie欺骗, 所以它是不安全的。

session存储在服务器上, 对客户端是透明的, 不存在敏感信息泄漏的风险。

4. 服务器压力不同

cookie保管在客户端, 不占用服务器资源。对于并发用户十分多的网站, cookie是很好的选择。

session是保管在服务器端的，每个用户都会产生一个session。假如并发访问的用户十分多，会产生十分多的session，耗费大量的内存。

2. 客户端禁用cookie怎么办？这种实现方式安全吗？

session的实现方式有两种。

第一种：通过cookies实现。如果浏览器支持cookies，创建session的时候会把sessionID放在cookies里面。

第二种：通过重写URL。如果浏览器不支持cookies，可以自己编程使用URL重写的方式实现session（访问页面的时候在地址栏里面，URL后会跟上**sessionID**）。

不安全，sessionID会暴露。

3. Session的生命周期

Session的生命周期：

创建：

sessionid第一次产生是在直到某server端程序调用 `HttpServletRequest.getSession(true)`这样的语句时才被创建。

删除：

- 服务器会把长时间没有活动的Session从服务器内存中清除，此时Session便失效。可以设置session超时时间
- 程序调用`HttpSession.invalidate()`

4. SessionID有什么用

那么Session在何时创建呢？当然还是在服务器端程序运行的过程中创建的，不同语言实现的应用程序有不同创建Session的方法，而在Java中是通过调用`HttpServletRequest`的`getSession`方法（使用true作为参数）创建的。在创建了Session的同时，服务器会为该Session生成唯一的Session id，而这个Session id在随后的请求中会被用来重新获得已经创建的Session；在Session被创建之后，就可以调用Session相关的方法往Session中增加内容了，而这些内容只会保存在服务器中，发到客户端的只有**Session id**；当客户端再次发送请求的时候，会将这个**Session id**带上，服务器接受到请求之后就会依据**Session id**找到相应的Session，从而再次使用之。

5. 怎样存储海量Session

6. Session共享方案

2. Servlet的生命周期

init() 方法

init 方法被设计成只调用一次。它在第一次创建 Servlet 时被调用，用于 Servlet 的初始化，初始化的数据，可以在整个生命周期中使用。

service() 方法

service() 方法是执行实际任务的主要方法。Servlet 容器（Tomcat、Jetty 等）调用 service() 方法来处理来自客户端（浏览器）的请求，并把相应结果返回给客户端。

每次 Servlet 容器接收到一个 Http 请求，Servlet 容器会产生一个新的线程并调用 Servlet 实例的 service 方法。service 方法会检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut、doDelete 方法。所以，在编码请求处理逻辑的时候，我们只需要关注 doGet()、或 doPost() 的具体实现即可。

destroy() 方法

destroy() 方法也只会被调用一次，在 Servlet 生命周期结束时调用。destroy() 方法主要用来清扫“战场”，执行如关闭数据库连接、释放资源等行为。

调用 destroy 方法之后，servlet 对象被标记为垃圾回收，等待 JVM 的垃圾回收器进行处理。

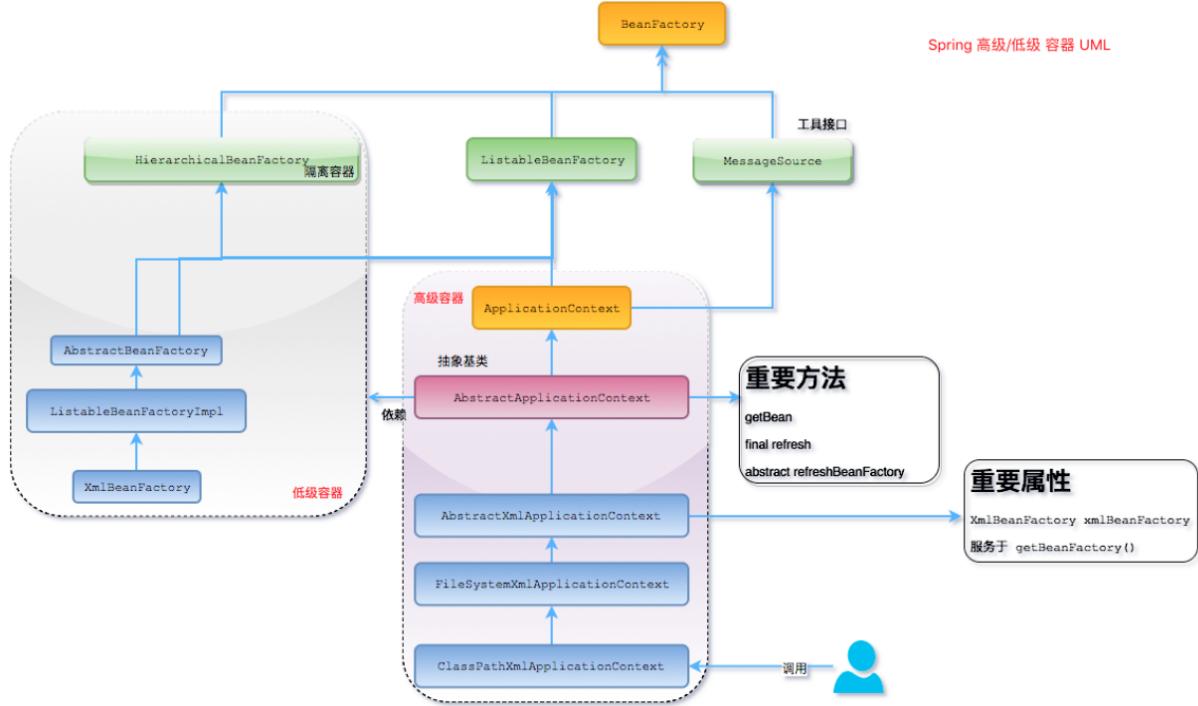
002 SPRING

1. IoC

IoC (Inversion of Control) / DI (Dependency Injection) 控制反转/依赖注入，是一种编程思想。

1. 编码流程：

- (1) applicationContext.xml 文件—通过 bean 标签，通过工厂 getBean() 方法
- (2) applicationContext 接口，实例化 ClassPathXmlApplicationContext
- (3) 获取对象实例



SpringIOC加载全过程

https://blog.csdn.net/qq_34203492/article/details/83865450

<https://www.cnblogs.com/stateis0/p/9779011.htm>

IoC底层实现原理

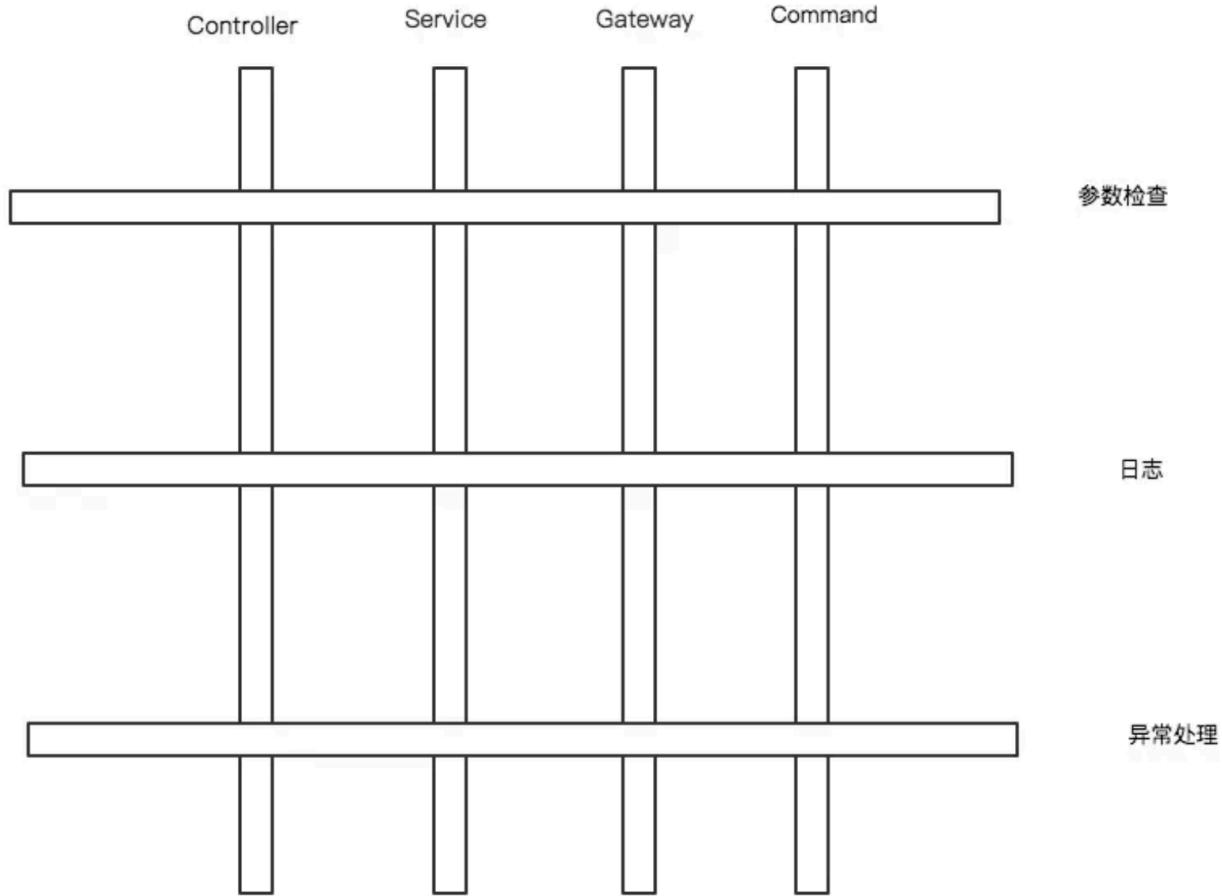
反射

2. AOP

1. 实际应用

AOP (Aspect Oriented Programming) 是能够让我们在不影响原有功能的前提下，为软件横向扩展功能。那么横向扩展怎么理解呢，我们在WEB项目开发中，通常都遵守三层原则，包括控制层

(Controller) ->业务层 (Service) ->数据层 (dao) ,那么从这个结构下来的为纵向，它具体的某一层就是我们所说的横向。我们的AOP就是可以作用于这某一个横向模块当中的所有方法。



Aspect (切面) : Aspect 声明类似于 Java 中的类声明，在 Aspect 中会包含着一些 Pointcut 以及相应的 Advice。

Joint point (连接点) : 表示在程序中明确定义的点，典型的包括方法调用，对类成员的访问以及异常处理程序块的执行等等，它自身还可以嵌套其它 joint point。

Pointcut (切点) : 表示一组 joint point，这些 joint point 或是通过逻辑关系组合起来，或是通过通配、正则表达式等方式集中起来，它定义了相应的 Advice 将要发生的地方。

Advice (增强) : Advice 定义了在 Pointcut 里面定义的程序点具体要做的操作，它通过 before、after 和 around 来区别是在每个 joint point 之前、之后还是代替执行的代码。

Target (目标对象) : 织入 Advice 的目标对象。

Weaving (织入) : 将 Aspect 和其他对象连接起来，并创建 Adviced object 的过程

2. 实现原理【动态代理】

Spring AOP的底层实现原理是动态代理

1. 相对于静态代理，动态代理的优势

静态代理与动态代理的区别主要在：

- 静态代理在编译时就已经实现，编译完成后代理类是一个实际的class文件

- 动态代理是在运行时动态生成的，即编译完成后没有实际的class文件，而是在运行时动态生成类字节码，并加载到JVM中

2. 反射和动态代理有什么联系

3. JDK动态代理和Cglib动态代理的区别

cglib与动态代理最大的区别就是

- 使用动态代理的对象必须实现一个或多个接口
- 使用cglib代理的对象则无需实现接口，达到代理类无侵入。

3. bean

1. 什么是Spring中的bean

Spring Bean是被实例的，组装的及被Spring 容器管理的Java对象。

1. 装配bean方式，优点和缺点是什么

而在 Spring 中提供了 3 种方法进行配置：

- 在 XML 文件中显式配置
- 在 Java 的接口和类中实现配置
- 隐式 Bean 的发现机制和自动装配原则

1. XML文件配置

2. 通过注解方式@Component+@ComponentScan

在 Spring 中，它提供了两种方式来让 Spring IoC 容器发现 bean：

- 组件扫描：通过定义资源的方式，让 Spring IoC 容器扫描对应的包，从而把 bean 装配进来。
- 自动装配：通过注解定义，使得一些依赖关系可以通过注解完成。
- 明显的弊端：
 - 对于 `@ComponentScan` 注解，它只是扫描所在包的 Java 类，但是更多的时候我们希望的是可以扫描我们指定的类
 - 上面的例子只是注入了一些简单的值，测试发现，通过 `@Value` 注解并不能注入对象

3. @Autowired自动装配【最常用方式】

接口+实现类

参考：<https://www.cnblogs.com/wmayskxz/p/8830632.html>

4. @Resource和@Autowired

1. 只有一个实现类的时候

@Autowired和@Resource效果相同

2. 同时存在多个实现类的时候

比如Human接口有两个实现类Man | Human，需要指定实现类

```
//1. 方式一
@Resource
@Qualifier("woman")
private Human human;

//2. 方式二
@Resource(name="woman")
private Human human;
```

可以使用@Primary注解和@Autowired注解配合使用

```
@Service
@Primary
public class Man implements Human {

    public String runMarathon() {
        return "A man run marathon";
    }
}
```

5. Spring核心组件

1.Bean

Spring是面向Bean的编程，Bean是Spring的主角

前面介绍了 Spring 的三个核心组件，如果再在它们三个中选出核心的话，那就非 Beans 组件莫属了，为何这样说，其实 Spring 就是面向 Bean 的编程（BOP,Bean Oriented Programming），Bean 在 Spring 中才是真正主角。Bean 在 Spring 中作用就像 Object 对 OOP 的意义一样，没有对象的概念就像没有面向对象编程，Spring 中没有 Bean 也就没有 Spring 存在的意义。

Spring解决的问题便是把对象间的以来关系转而用配置文件来管理，也就是它的依赖注入机制。而这个注入关系在一个叫做IOC中的容器中进行管理，那么在IOC容器中就是被Bean包裹的对象，Spring正是通过把对象包装在Bean中从而达到管理这些对象及做一系列额外操作的目的。

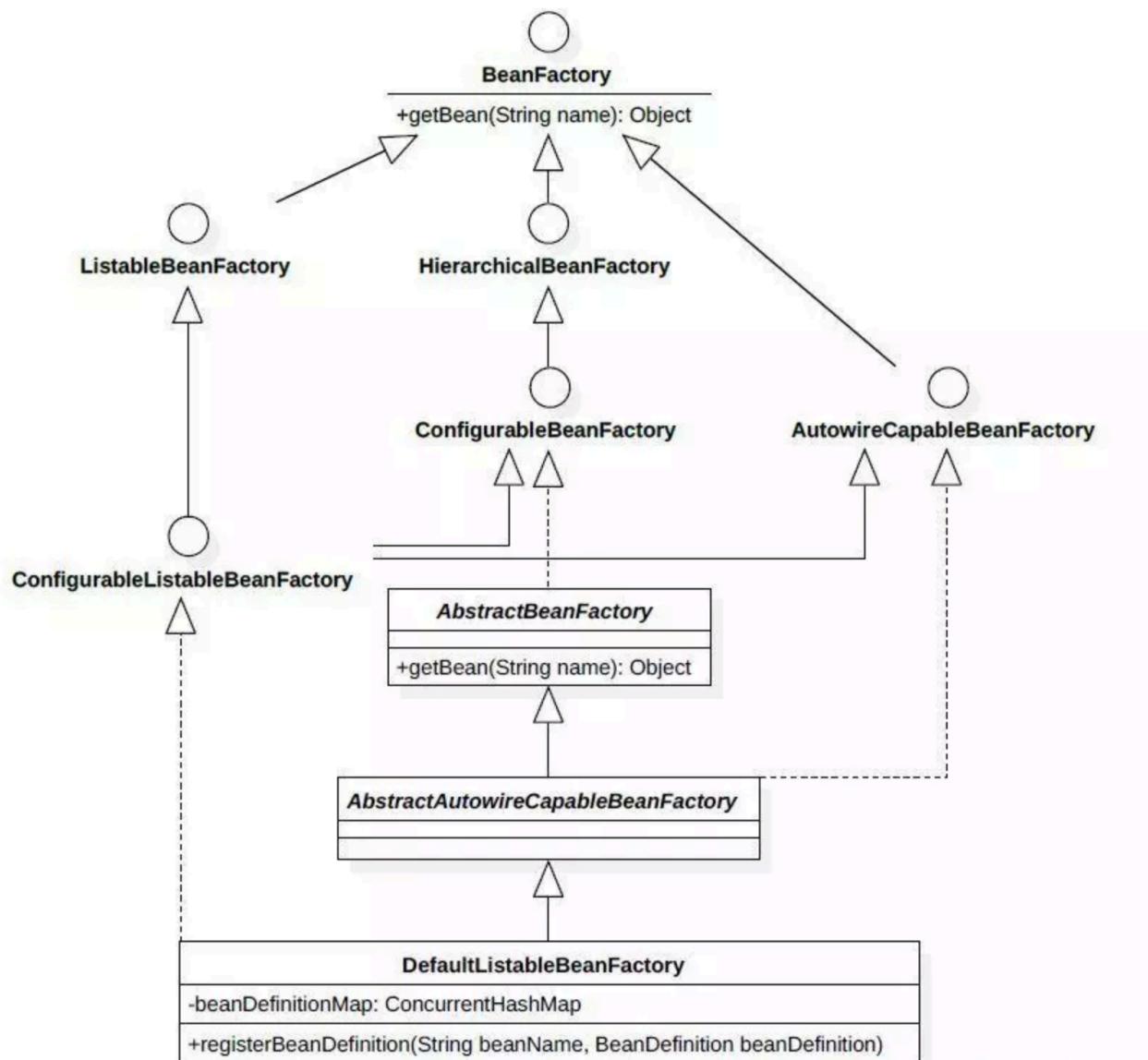
Bean组件位于Spring的org.springframework.beans包下，在这个包下的所有类主要解决3件事：

1.Bean的定义，2.Bean的创建，3.Bean的解析。对Spring的使用者来说，我们唯一需要关心的就是Bean的创建，其他两个由Spring帮我们完成。

SpringBean的创建是典型的工厂模式，顶级接口是BeanFactory。BeanFactory有三个子类，

1.ListableBeanFactory 2.HierarchicalBeanFactory AutowireCapableBeanFactory。但是最终实现类是DefaultListableBeanFactory，它实现了所有的接口。这些接口都有自己使用的场合，主要是为了区分在Spring内部对象的传递和转化过程中，对对象的数据访问所做的限制。ListableBeanFactory 表示 Bean是可列表的，HierarchicalBeanFactory表示这些Bean是可继承的，即表示每个Bean都有可能是有父类的，AutoWireCapableBeanFactory接口定义Bean的自动装配规则。这四个接口定义了Bean的

集合，Bean之间的关系和行为。



BeanFactory类结构图

Bean的定义完整地描述在Spring的配置文件中节点的所有信息，只要成功定义一个节点后，在Spring的内部它就被转换为BeanDefinition对象，以后所有的操作都是围绕着这个对象进行的。

2.Core

前面把Bean比作一场演出中的演员，Context就是这场演出的舞台背景，而Core就是演出的道具。

Core组件作为Spring的核心组件，包含了很多关键类。一个重要的组成部分就是定义了资源的访问方式，把所有资源都抽象成了一个Resource接口，对使用者屏蔽了资源类型。同理把所有的资源加载者都抽象成了一个ResourceLoader接口，又屏蔽了资源加载者的差异，默认实现是DefaultResourceLoader。

3.Context

Context在Spring的org.springframework.context包下，前面已经讲解了Context组件在Spring中的作用，它其实就是给Spring提供一个运行时环境，用以保存各个对象的状态。

ApplicationContext是**Context**的顶级父类，它除了能标识一个应用环境的基本信息之外，还继承了五个接口。这五个接口主要是扩展了**Context**的功能。**ApplicationContext**继承了**BeanFactory**和**ResourceLoader**接口，使得**ApplicationContext**可以接触到任何外部资源。**ApplicationContext**的子类主要包含两个方面：

- (1) **ConfigurableApplicationContext**表示该**Context**是可以修改的。最常使用的是可更新的**Context**，即**AbstractRefreshableApplicationContext**类。
- (2) **WebApplicationContext**为Web准备的**Context**，它可以直接访问**ServletContext**，但用得极少。

ApplicationContext需要完成下面几件事：

- (1) 标识一个应用环境
- (2) 利用**BeanFactory**建立**Bean**对象
- (3) 保存对象关系表
- (4) 能捕获各种事件

Context作为Spring的IOC容器，基本上整合了Spring的大部分功能，或者说是大部分功能的基础。

核心组件如何协同工作

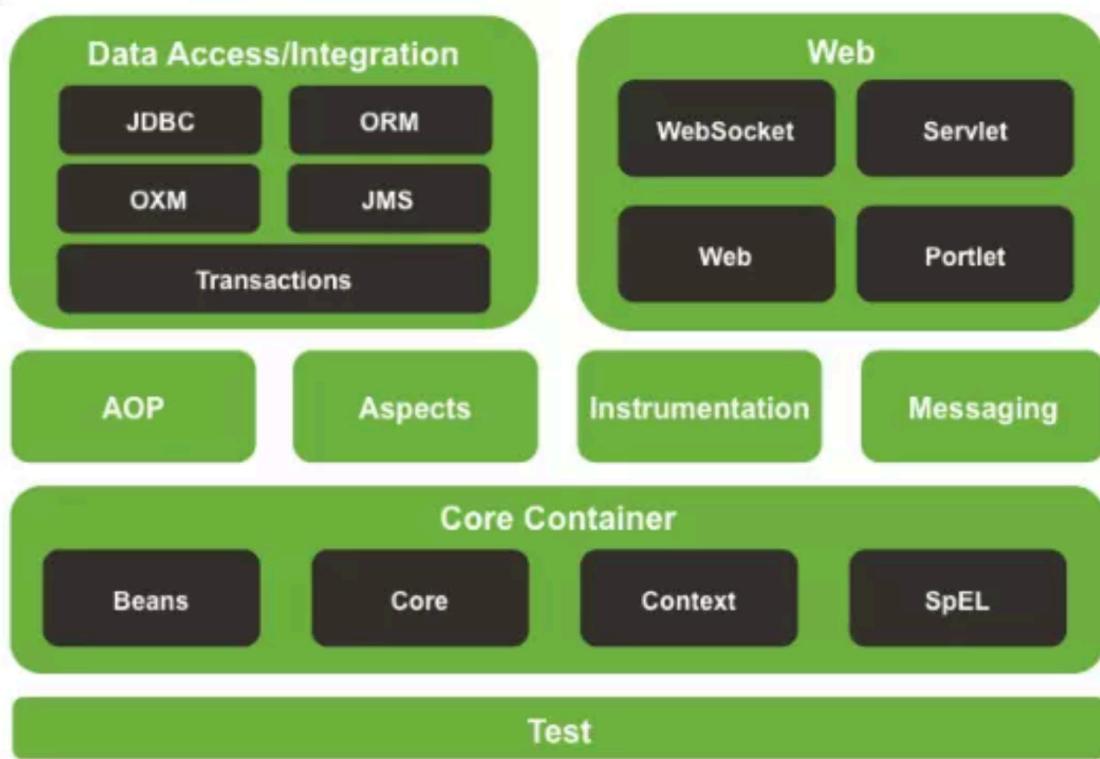
Bean封装的是object，object必然有数据，如何给这些数据提供生存环境就是**Context**要解决的问题，对**Context**来说就是它要发现每个Bean之间的关系，为它建立这种关系并且维持好这种关系。**Context**就像是Bean关系的集合，这个集合叫做IOC容器。Core组件就是发现，建立，和维护每个Bean之间关系所需要的一系列工具，从这个角度看，把Core组件叫做Util更能让人理解。

6. 谈谈你对Spring的理解

- SpringFramework
- SpringData
- SpringSecurity
- SpringBoot
- SpringCloud

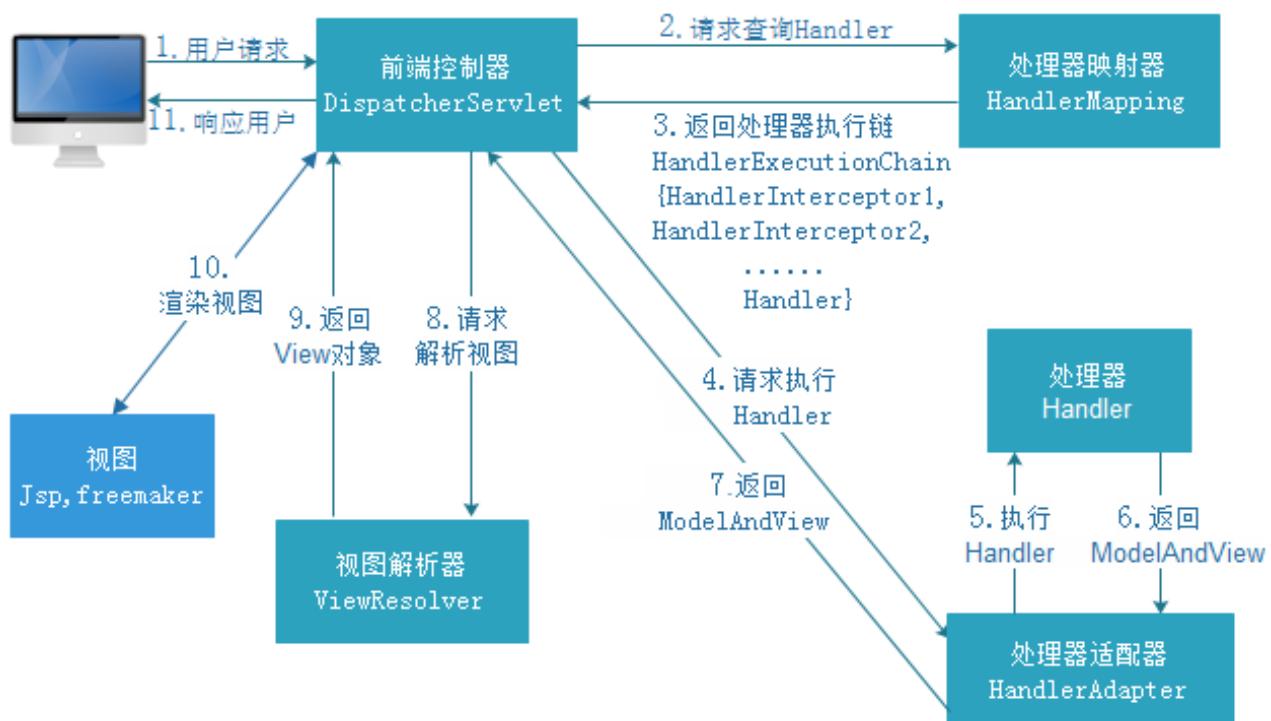


Spring Framework Runtime



003 SpringMVC

SpringMVC的执行流程 ★★★★



- 一个请求匹配前端控制器 DispatcherServlet 的请求映射路径(在 web.xml中指定), WEB 容器将该请求转交给 DispatcherServlet 处理
- DispatcherServlet 接收到请求后, 将根据 请求信息 交给 处理器映射器 (HandlerMapping)
- HandlerMapping 根据用户的url请求 查找匹配该url的 Handler, 并返回一个执行链
- DispatcherServlet 再请求 处理器适配器(HandlerAdapter) 调用相应的 Handler 进行处理并返回 ModelAndView 给 DispatcherServlet
- DispatcherServlet 将 ModelAndView 请求 ViewReslover (视图解析器) 解析, 返回具体 View
- DispatcherServlet 对 View 进行渲染视图 (即将模型数据填充至视图中)
- DispatcherServlet 将页面响应给用户

DispatcherServlet: 前端控制器

用户请求到达前端控制器, 它就相当于mvc模式中的c, dispatcherServlet是整个流程控制的中心, 由它调用其它组件处理用户的请求, dispatcherServlet的存在降低了组件之间的耦合性。

HandlerMapping: 处理器映射器

HandlerMapping负责根据用户请求url找到Handler即处理器, springmvc提供了不同的映射器实现不同的映射方式, 例如: 配置文件方式, 实现接口方式, 注解方式等。

Handler: 处理器

Handler 是继DispatcherServlet前端控制器的后端控制器, 在DispatcherServlet的控制下 Handler对具体的用户请求进行处理。由于Handler涉及到具体的用户业务请求, 所以一般情况需要程序员根据业务需求开发Handler。

HandlerAdapter: 处理器适配器

通过HandlerAdapter对处理器进行执行, 这是适配器模式的应用, 通过扩展适配器可以对更多类型的处理器进行执行。

ViewResolver: 视图解析器

View Resolver负责将处理结果生成View视图, View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址, 再生成View视图对象, 最后对view进行渲染将处理结果通过页面展示给用户。

View: 视图

springmvc框架提供了很多的view视图类型的支持, 包括: jstlView、freemarkerView、pdfView 等。我们最常用的视图就是jsp。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户, 需要由程序员根据业务需求开发具体的页面。

5. Spring是如何进行事务管理的。

6. Spring数据访问原理是什么。

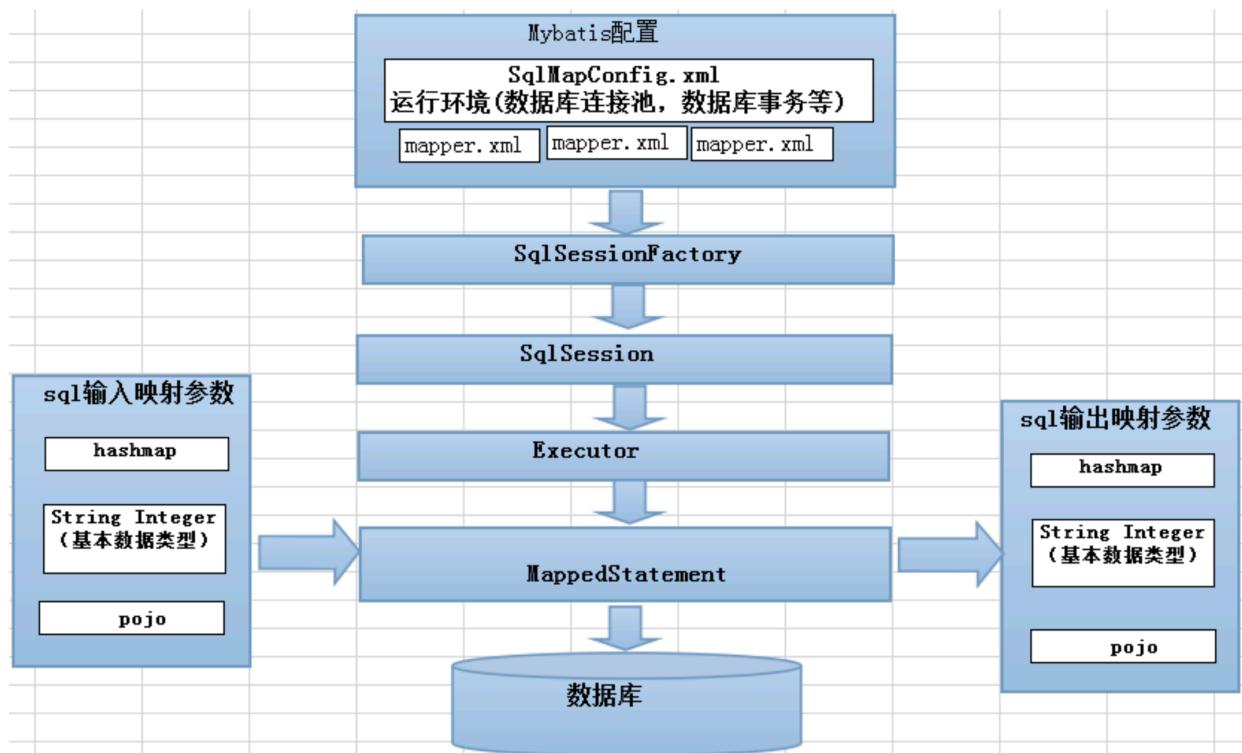
Spring 事务管理

004 Mybatis

1. 为什么要使用MyBatis

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注SQL本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

Mybatis通过xml或注解的方式将要执行的各种statement (statement、preparedStatemnt、CallableStatement) 配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。



2.

1. mybatis和hibernate的不同点，mybatis和jdbc的相同点和不同点

2. #{} 和\${}的不同

3. 谈一谈动态sql

4. 延迟加载及缓存问题

0x 中间件

001 Redis

1. Redis有哪些基本数据类型

1. string类型

- 这是最简单的类型，就是普通的 set 和 get，做简单的 KV 缓存。
 - `set name mio`
 - `get name mio`
- 应用：微博粉丝数

2. hash类型

- 这个是类似 map 的一种结构，这个一般就是可以将**结构化的数据**，比如一个对象（前提是这个对象没嵌套其他的对象）给缓存在 redis 里，然后每次读写缓存的时候，可以就操作 hash 里的某个**字段**。

- `hset person name bingo`
- `hset person age 20`
- `hget person name`

- 应用：存储用户信息

3. List类型

- 有序列表，可以从左边和右边插入

- `lpush mylist 1`
- `lpush mylist 2 3`
- `rpush mylist 0`
- `lrange mylist 0 4`

- 应用：消息队列

4. Set类型

- set 是无序集合，自动去重。
- 为什么不直接用hashset去重

- 直接基于 set 将系统里需要去重的数据扔进去，自动就给去重了，如果你需要对一些数据进行快速的全局去重，你当然也可以基于 jvm 内存里的 HashSet 进行去重，但是如果你的某个系统部署在多台机器上呢？得基于 redis 进行全局的 set 去重。
- 可以基于 set 玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁？对吧。
- `sadd mySet 1`
- `sinter mySet yourSet` 求交集

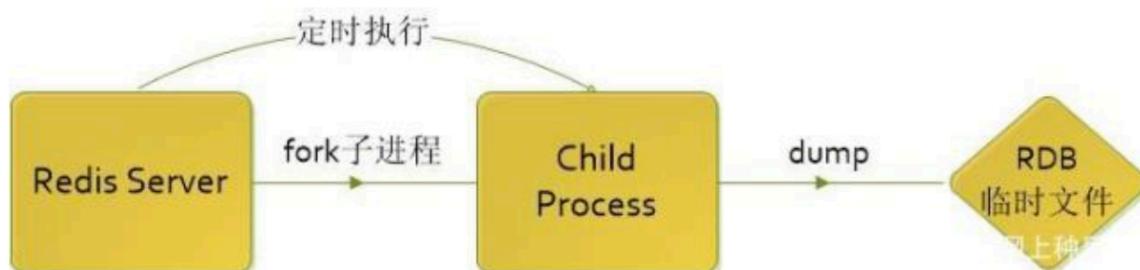
5. Sorted Set类型

- sorted set 是排序的 set，去重但可以排序，写进去的时候给一个分数，自动根据分数排序。
 - `zadd board 90 mio`
 - `zadd board 80 mio2`
 - `zrevrange board 0 3`
- 应用：列出前100名畅销的商品

2. Redis如何持久化? ★★★★★

1. RDB方式

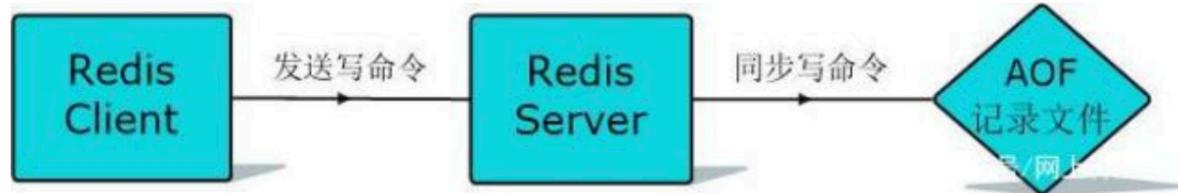
- RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。RDB是Redis默认的持久化方式，会在对应的目录下生产一个dump.rdb文件，重启会通过加载dump.rdb文件恢复数据。
-



- 优点
 - 只有一个文件**dump.rdb**，方便持久化；
 - 性能最大化，**fork子进程来完成写操作**，让主进程继续处理命令，所以是IO最大化（使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能）；
 - 如果数据集偏大，RDB的启动效率会比AOF更高。
- 缺点
 - 数据安全性低。（**RDB是间隔一段时间进行持久化，如果持久化之间redis发生故障，会发生数据丢失**。所以这种方式更适合数据要求不是特别严格的时候）
 - 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

2. AOF方式

- AOF持久化是以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，文件中可以看到详细的操作记录。她的出现是为了弥补RDB的不足（数据的不一致性），所以它采用日志的形式来记录每个写操作，并追加到文件中。Redis重启的会根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。



- 优点

- 数据安全性更高，AOF持久化可以配置appendfsync属性，其中always，每进行一次命令操作就记录到AOF文件中一次。
- 通过append模式写文件，即使中途服务器宕机，可以通过redis-check-aof工具解决数据一致性问题。

- 缺点

- AOF文件比RDB文件大，且恢复速度慢；数据集大的时候，比rdb启动效率低。
- 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。

3. Redis单线程模型 ★

- 单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。
- Redis为什么这么快
 - (1) 绝大部分请求是纯粹的内存操作（非常快速）
 - (2) 采用单线程，避免了不必要的上下文切换和竞争条件
 - (3) 非阻塞IO - IO多路复用

4. 缓存穿透、缓存雪崩区别和解决方案 ★★★★☆

1. 缓存穿透

- 缓存击穿表示恶意用户模拟请求很多缓存中不存在的数据，由于缓存中都没有，导致这些请求短时间内直接落在了数据库上，导致数据库异常。
 - 缓存失效，直接访问数据库
- 解决方案：布隆过滤器
 - 使用布隆过滤器判断key是否存在[能够判定一个KEY一定不存在]，不存在的KEY就不用查询数据库，直接丢弃请求

2. 缓存雪崩

- 如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。
- 解决方案
 1. 分布均匀：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

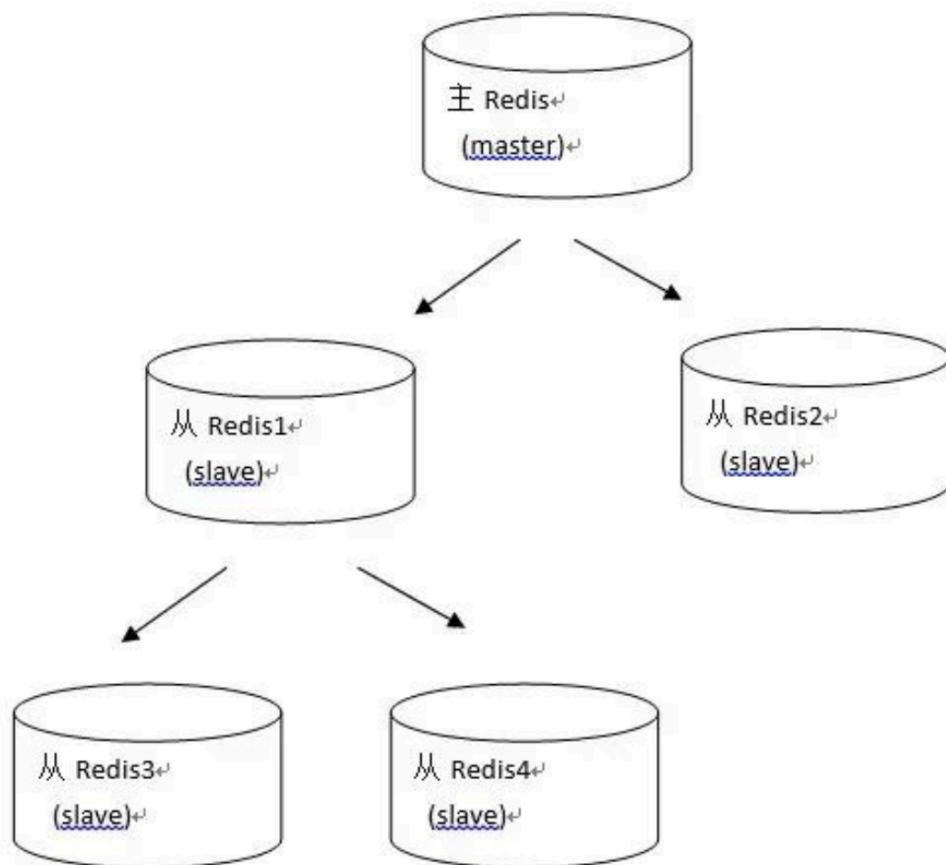
2. 限制流量：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量

5. 主从复制

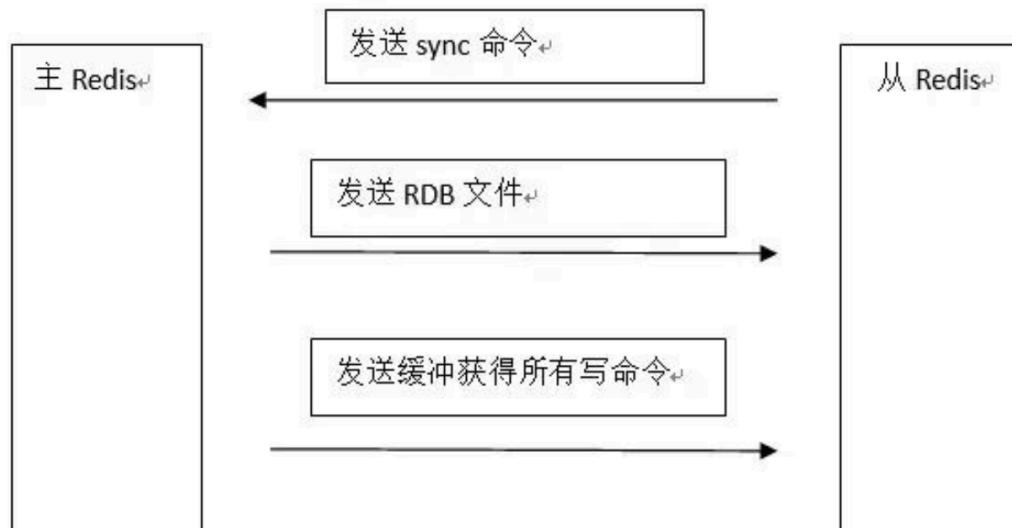
- 解决了什么问题

- 持久化保证了即使 redis 服务重启也会丢失数据，因为 redis 服务重启后会将硬盘上持久化的数据恢复到内存中，但是当 redis 服务器的硬盘损坏了可能会导致数据丢失，如果通过 redis 的主从复制机制就可以避免这种单点故障

-



- 复制过程



- slave 服务启动，slave 会建立和 master 的连接，发送 sync 命令

- master 启动一个后台进程将数据库快照保存到 RDB 文件中

6. 缓存淘汰策略

1. 缓存过期策略

主要从CPU使用和内存占用的角度来分析

Redis同时使用了惰性删除与定期删除。

- 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
- 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限制耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。（expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。）

2. 内存淘汰策略

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

- **noeviction**: 当内存不足以容纳新写入数据时，新写入操作会报错。
- **allkeys-lru**: 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。
- **allkeys-random**: 当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。
- **volatile-lru**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。
- **volatile-random**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。
- **volatile-ttl**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

7. String数据类型实现原理

String的数据类型是由**SDS (Simple Dynamic String)** 实现的。Redis并没有采用C语言的字符串表示，而是自己构建了一种名为SDS的抽象类型，并将SDS作为Redis的默认字符串表示。

```
redis>SET msg "hello world"  
OK
```

上边设置key=msg, value=hello world的键值对，它们的底层存储是：键（key）是字符串类型，其底层实现是一个保存着“msg”的SDS。值（value）是字符串类型，其底层实现是一个保存着“hello world”的SDS。

注意：SDS除了用于实现字符串类型，还被用作AOF持久化时的缓冲区。

SDS的定义为：

```
struct sdshdr {  
    // buf 中已占用空间的长度  
    int len;  
  
    // buf 中剩余可用空间的长度  
    int free;  
  
    // 数据空间  
    char buf[];  
};
```

1. 直接获取字符串长度

时间复杂度：获取字符串长度 (SDS O (1))

我们一定会思考，redis为什么不使用C语言的字符串而是费事搞一个SDS呢，这是因为C语言用N+1的字符数组来表示长度为N的字符串，这样做在获取字符串长度，字符串扩展等操作方面效率较低，并且无法满足redis对字符串在安全性、效率以及功能方面的要求。

在C语言字符串中，为了获取一个字符串的长度，必须遍历整个字符串，时间复杂度为O(1)，而SDS中，有专门用于保存字符串长度的变量，所以可以在O (1) 时间内获得。

2. 防止缓冲区溢出

C字符串，容易导致缓冲区溢出，假设在程序中存在内存紧邻的字符串s1和s2，s1保存redis，s2保存MongoDB，如下图：

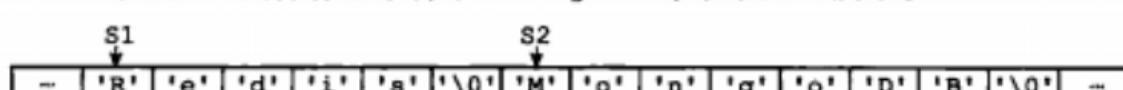


图 2-7 在内存中紧邻的两个 C 字符串 [net/u013679744](http://blog.csdn.net/u013679744)

如果我们现在将s1 的内容修改为redis cluster，但是又忘了重新为s1 分配足够的空间，这时候就会出现以下问题：

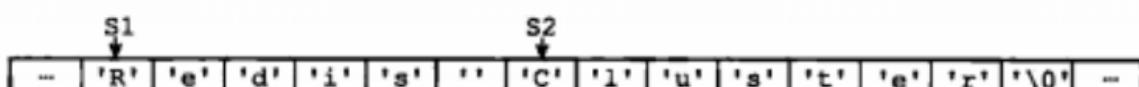


图 2-8 s1 的内容溢出到了 s2 所在的位置上 <http://blog.csdn.net/u013679744>

因为s1和s2是紧邻的，所以原本s2 中的内容已经被s1的内容给占领了，s2 现在为 cluster，而不是“MongoDb”。而Redis中的SDS就杜绝了发生缓冲区溢出的可能性。当我们需要对一个SDS 进行修改的时候，redis 会在执行拼接操作之前，预先检查给定SDS 空间是否足够 (free记录了剩余可用的数据长度)，如果不夠，会先拓展SDS 的空间，然后再执行拼接操作。

3. 减少扩展或收缩字符串带来的内存重分配次数

当字符串进行扩展或收缩时，都会对内存空间进行重新分配。

1. 字符串拼接会产生字符串的内存空间的扩充，在拼接的过程中，原来的字符串的大小很可能小于拼接后的字符串的大小，那么这样的话，就会导致一旦忘记申请分配空间，就会导致内存的溢出。
2. 字符串在进行收缩的时候，内存空间会相应的收缩，而如果在进行字符串的切割的时候，没有对内存的空间进行一个重新分配，那么这部分多出来的空间就成为了内存泄露。

比如：字符串"redis"，当进行字符串拼接时，将redis+cluster=13，会将SDS的长度修改为13，同时将free也改为13，这意味着进行预分配了，将buffer大小变为了26。这是为了如果再次执行字符串拼接操作，如果拼接的字符串长度<13,就不需要重新进行内存分配了。

通过这种预分配策略，SDS将连续增长N次字符串所需的内存重分配次数从必定N次降低为最多N次。通过惰性空间释放，SDS避免了缩短字符串时所需的内存重分配操作，并未将来可能有的增长操作提供了优化。

4. 二进制安全

C字符串中的字符必须符合某种编码，并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符将被误认为是字符串结尾，这些限制使得C字符串只能保存文本数据，而不能保存想图片，音频，视频，压缩文件这样的二进制数据。

但是在Redis中，不是靠空字符来判断字符串的结束的，而是通过len这个属性。那么，即便是中间出现了空字符对于SDS来说，读取该字符仍然是可以的。但是，SDS依然可以兼容部分C字符串函数。

8. 哨兵机制

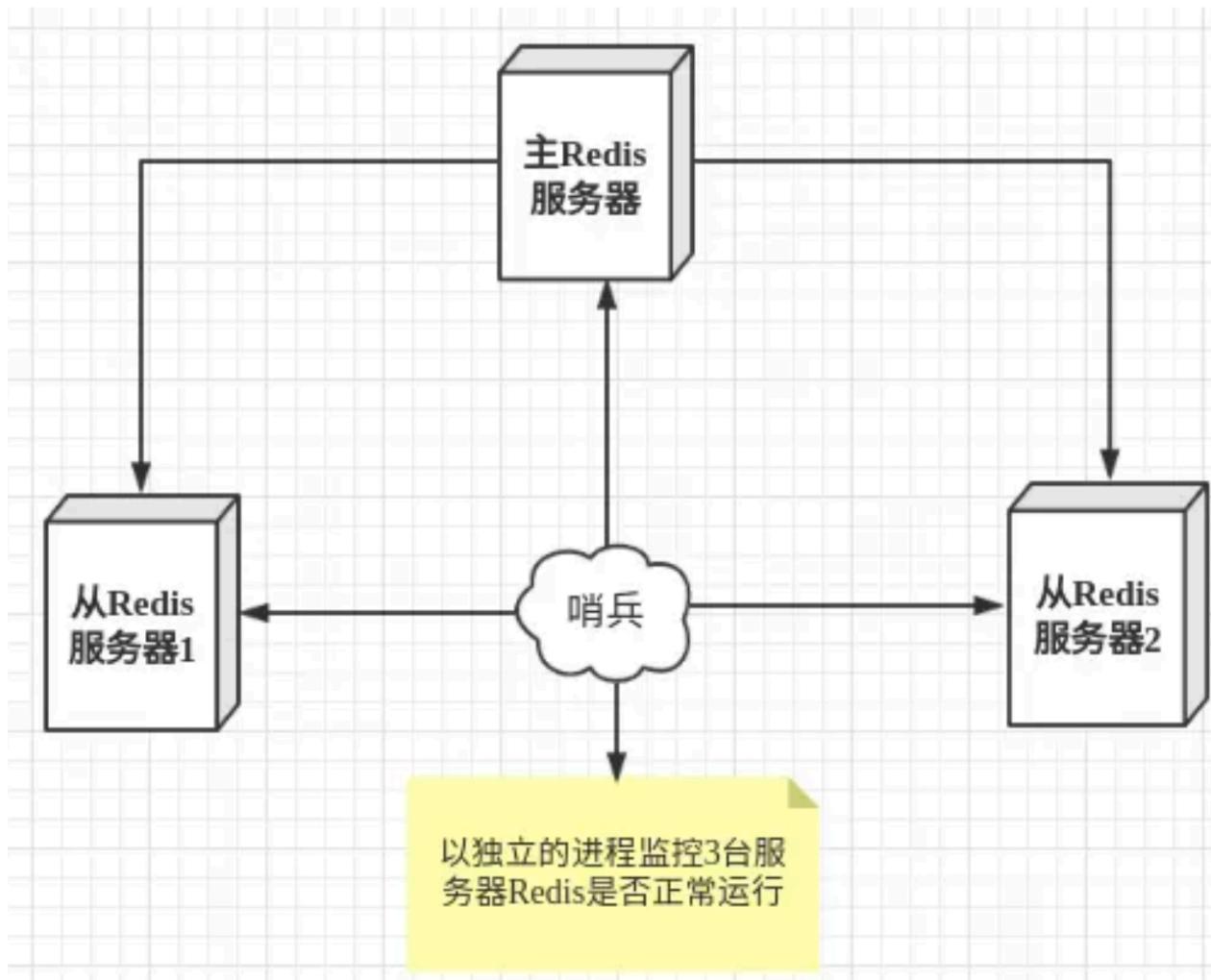
1. 哨兵Sentinel的意义

Redis和MySQL一样，可以使用主从备份模式。

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，更多时候，我们优先考虑哨兵模式。哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。

这里的哨兵有两个作用

- 通过发送命令，让Redis服务器返回监控其运行状态，包括主服务器和从服务器。
- 当哨兵监测到master宕机，会自动将slave切换成master，然后通过发布订阅模式通知其他的从服务器，修改配置文件，让它们切换主机。



2. 多哨兵模式

然而一个哨兵进程对Redis服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

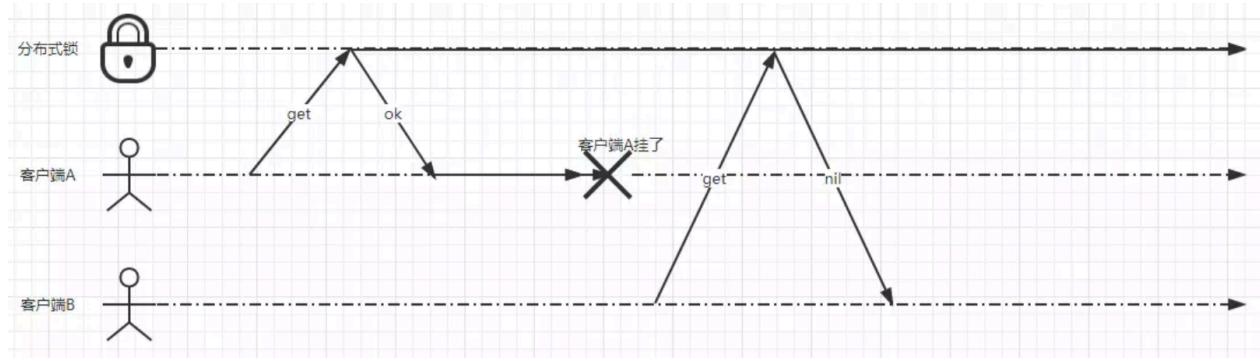
用文字描述一下故障切换（failover）的过程。假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为主服务器不可用，这个现象成为**主观下线**。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行failover操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为**客观下线**。这样对于客户端而言，一切都是透明的。

9. 分布式锁

1. 使用场景

分布式锁是控制分布式系统或不同系统之间共同访问共享资源的一种锁实现，如果不同的系统或同一个系统的不同主机之间共享了某个资源时，往往需要互斥来防止彼此干扰来保证一致性。

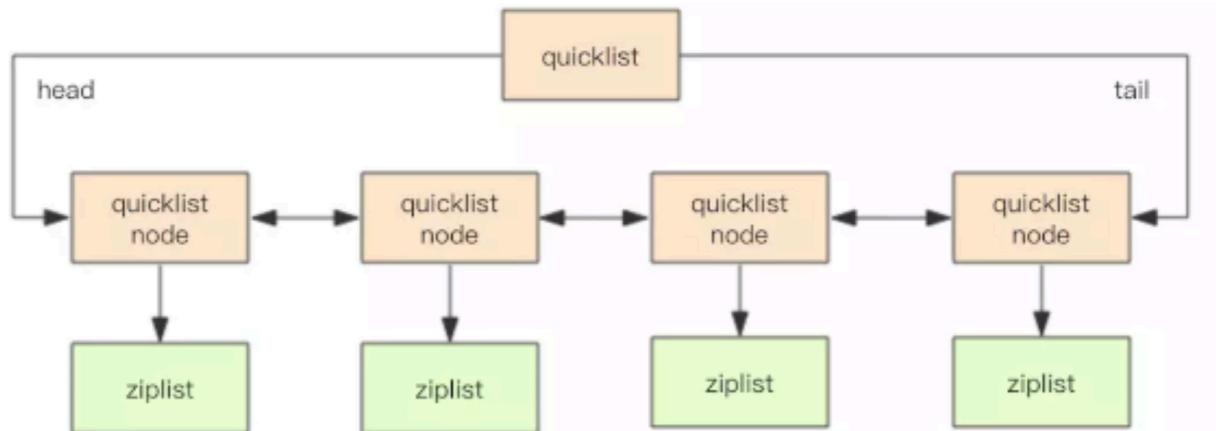
2. 实现



Ref: <https://juejin.im/post/5b737b9b518825613d3894f4>

10. List数据类型实现原理

1. 为什么不是简单的双向链表



quicklist的每个节点都是一个ziplist。ziplist我们已经在上一篇介绍过。ziplist本身也是一个能维持数据项先后顺序的列表（按插入位置），而且是一个内存紧缩的列表（各个数据项在内存上前后相邻）。比如，一个包含3个节点的quicklist，如果每个节点的ziplist又包含4个数据项，那么对外表现上，这个list就总共包含12个数据项。

- 双向链表便于在表的两端进行push和pop操作，但是它的内存开销比较大。首先，它在每个节点上除了要保存数据之外，还要额外保存两个指针；其次，双向链表的各个节点是单独的内存块，地址不连续，节点多了容易产生内存碎片。【考虑到链表的附加空间相对太高，prev 和 next 指针就要占去 16 个字节 (64bit 系统的指针是 8 个字节)，另外每个节点的内存都是单独分配，会加剧内存的碎片化，影响内存管理效率。】
- ziplist由于是一整块连续内存，所以存储效率很高。但是，它不利于修改操作，每次数据变动都会引发一次内存的realloc。特别是当ziplist长度很长的时候，一次realloc可能会导致大批量的数据拷贝，进一步降低性能。

2. 底层细节

<https://blog.csdn.net/zhaoliang831214/article/details/82054476>

<https://www.cnblogs.com/virgosnail/p/9542470.html>

002 RabbitMQ

1. 你的项目中为什么要使用RabbitMQ，而不是其他的消息队列



特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低；管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

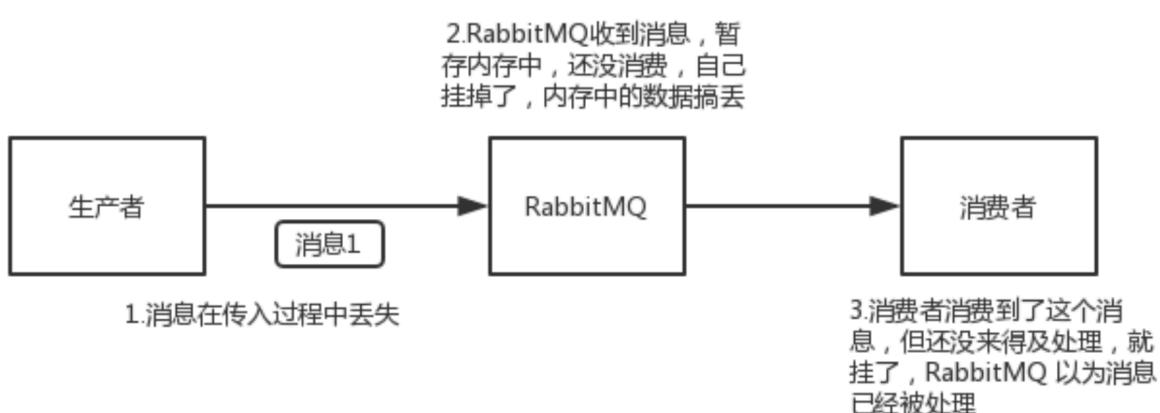
(1) 中小型软件公司，建议选RabbitMQ。一方面，erlang语言天生具备高并发的特性，而且他的管理界面用起来十分方便。正所谓，成也萧何，败也萧何！他的弊端也在那里，虽然RabbitMQ是开源的，然而国内有几个能定制化开发erlang的程序员呢？所幸，RabbitMQ的社区十分活跃，可以解决开发过程中遇到的bug，这点对于中小型公司来说十分重要。不考虑rocketmq和kafka的原因是，一方面中小型软件公司不如互联网公司，数据量没那么大，选消息中间件，应首选功能比较完备的，所以kafka排除。不考虑rocketmq的原因是，rocketmq是阿里出品，如果阿里放弃维护rocketmq，中小型公司一般抽不出人来进行rocketmq的定制化开发，因此不推荐。

(2)大型软件公司，根据具体使用在rocketMq和kafka之间二选一。一方面，大型软件公司，具备足够的资金搭建分布式环境，也具备足够大的数据量。针对rocketMQ,大型软件公司也可以抽出人手对rocketMQ进行定制化开发，毕竟国内有能力改JAVA源码的人，还是相当多的。至于kafka，根据业务场景选择，如果有日志采集功能，肯定是首选kafka了。具体该选哪个，看使用场景。

2. 如何处理消息丢失 ★★★

- 消息可靠性
 - 没有重复数据
 - 数据不会丢失

RabbitMQ 消息丢失的 3 种情况

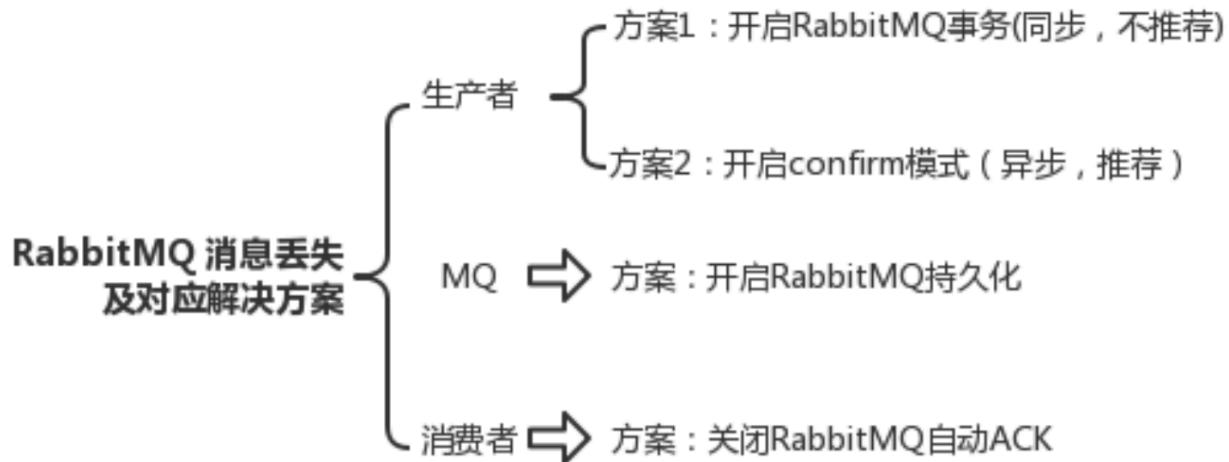


- 消息丢失的处理情况

- 生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 `channel.txSelect`，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 `channel.txRollback`，然后重试发送消息；如果收到了消息，那么可以提交事务 `channel.txCommit`。

- 就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，可能导致少量数据丢失，但是这个概率较小。
- RabbitMQ 如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。这个时候得用 RabbitMQ 提供的 `ack` 机制，简单来说，就是你关闭 RabbitMQ 的自动 `ack`，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack`？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



3. RabbitMQ的基本原理

1. 生产者、消费者和代理

在了解消息通讯之前首先要了解3个概念：生产者、消费者和代理。

生产者：消息的创建者，负责创建和推送数据到消息服务器；

消费者：消息的接收方，用于处理数据和确认消息；

代理：就是RabbitMQ本身，用于扮演“快递”的角色，本身不生产消息，只是扮演“快递”的角色。

2. 消息发送原理

首先你必须连接到Rabbit才能发布和消费消息，那怎么连接和发送消息的呢？

你的应用程序和Rabbit Server之间会创建一个TCP连接，一旦TCP打开，并通过了认证，认证就是你试图连接Rabbit之前发送的Rabbit服务器连接信息和用户名和密码，有点像程序连接数据库，使用Java有两种连接认证的方式，后面代码会详细介绍，一旦认证通过你的应用程序和Rabbit就创建了一条AMQP信道（Channel）。

信道是创建在“真实”TCP上的虚拟连接，AMQP命令都是通过信道发送出去的，每个信道都会有一个唯一的ID，不论是发布消息，订阅队列或者介绍消息都是通过信道完成的。

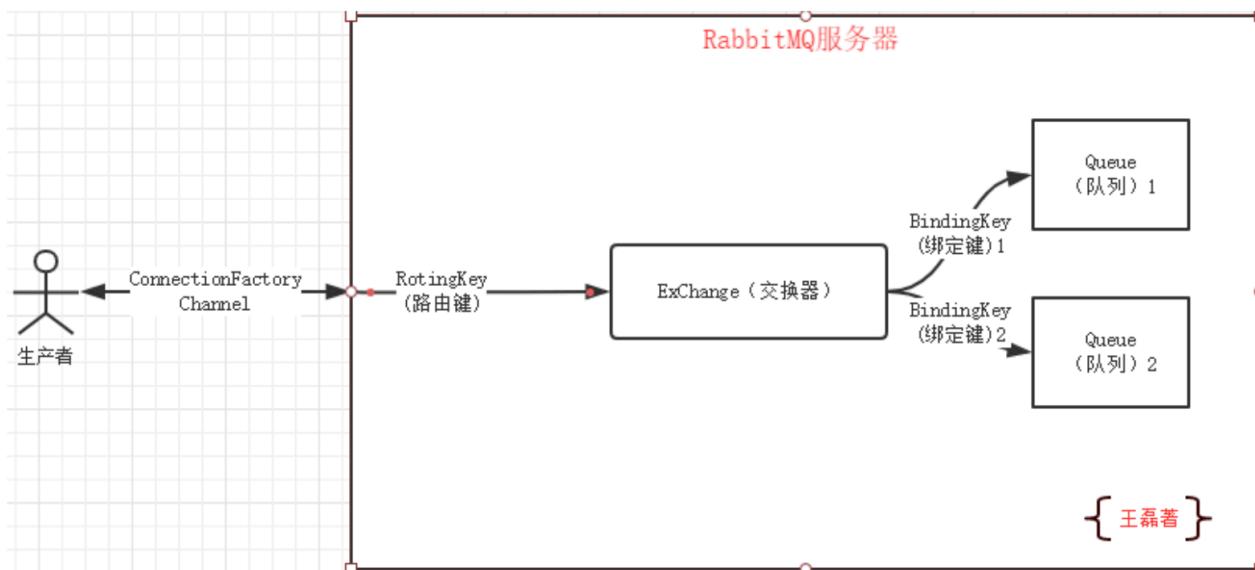
3. 为什么不通过TCP直接发送命令？

对于操作系统来说创建和销毁TCP会话是非常昂贵的开销，假设高峰期每秒有成千上万条连接，每个连接都要创建一条TCP会话，这就造成了TCP连接的巨大浪费，而且操作系统每秒能创建的TCP也是有限的，因此很快就会遇到系统瓶颈。

如果我们每个请求都使用一条TCP连接，既满足了性能的需要，又能确保每个连接的私密性，这就是引入信道【Channel】概念的原因。

- **ConnectionFactory**（连接管理器）：应用程序与Rabbit之间建立连接的管理器，程序代码中使用；
- **Channel**（信道）：消息推送使用的通道；
- **Exchange**（交换器）：用于接受、分配消息；

- **Queue (队列)** : 用于存储生产者的消息；
- **RoutingKey (路由键)** : 用于把生成者的数据分配到交换器上；
- **BindingKey (绑定键)** : 用于把交换器的消息绑定到队列上；



4. RabbitMQ中的持久化

当你把消息发送到Rabbit服务器的时候，你需要选择你是否要进行持久化，但这并不能保证Rabbit能从崩溃中恢复，想要Rabbit消息能恢复必须满足3个条件：

1. 投递消息的时候durable设置为true，消息持久化，代码：channel.queueDeclare(x, true, false, false, null)，参数2设置为true持久化；
2. 设置投递模式deliveryMode设置为2（持久），代码：channel.basicPublish(x, x, MessageProperties.PERSISTENT_TEXT_PLAIN,x)，参数3设置为存储纯文本到磁盘；
3. 消息已经到达持久化交换器上；
4. 消息已经到达持久化的队列；

持久化工作原理

Rabbit会将你的持久化消息写入磁盘上的持久化日志文件，等消息被消费之后，Rabbit会把这条消息标识为等待垃圾回收。

持久化的缺点

消息持久化的优点显而易见，但缺点也很明显，那就是性能，因为要写入硬盘要比写入内存性能较低很多，从而降低了服务器的吞吐量，尽管使用SSD硬盘可以使事情得到缓解，但他仍然吸干了Rabbit的性能，当消息成千上万条要写入磁盘的时候，性能是很低的。

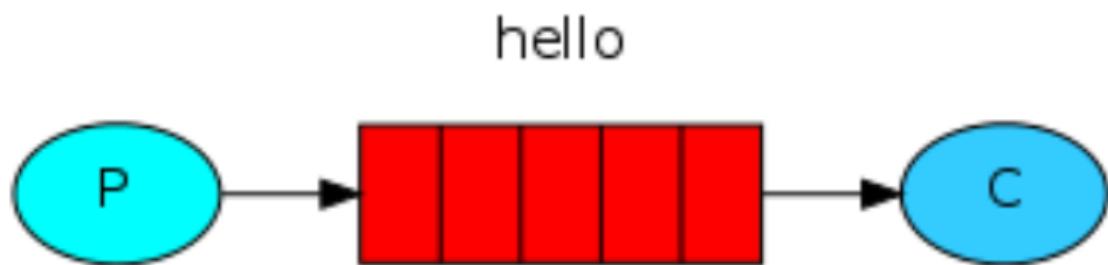
5. 如何保证一条消息不被多次消费 ★★★★

1. 什么时候会出现这种情况

2. 解决方案

6. RabbitMQ中的消息队列模型

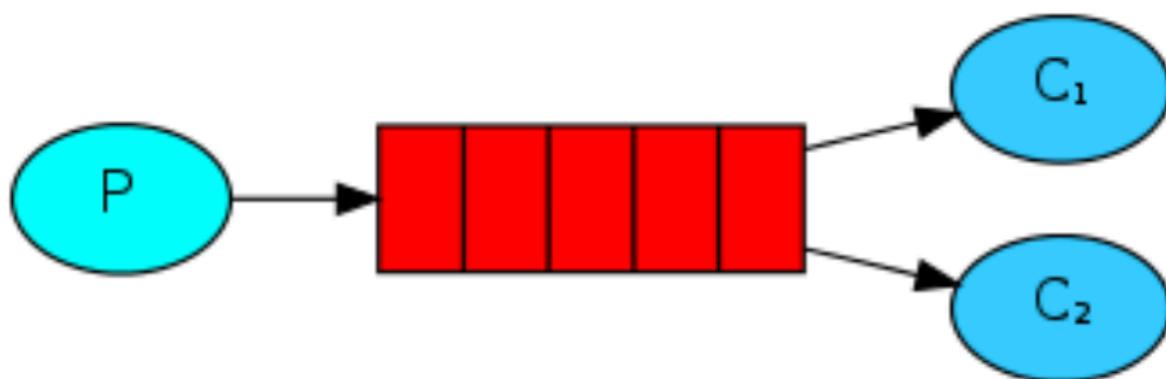
1. 单队列模型



2. 工作队列模型

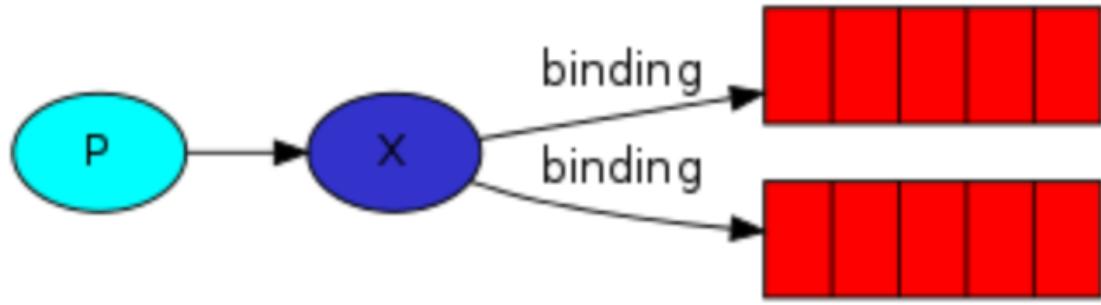
The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a *task* as a message and send it to the queue. A worker process running in the background will pop the tasks and eventually execute the job. **When you run many workers the tasks will be shared between them.**

一个生产者，多个消费者，将任务分配给不同的consumer



3. 发布/订阅模型

- 发布一次，消费多个。
- 举个用户注册的例子：用户在注册完后一般都会发送消息通知用户注册成功（失败）。如果在一个系统中，用户注册信息有邮箱、手机号，那么在注册完后会向邮箱和手机号都发送注册完成信息。利用MQ实现业务异步处理，如果是用工作队列的话，就会声明一个注册信息队列。注册完成之后生产者会向队列提交一条注册数据，消费者取出数据同时向邮箱以及手机号发送两条消息。但是实际上邮箱和手机号信息发送实际上是不同的业务逻辑，不应该放在一块处理。这个时候就可以利用发布/订阅模式将消息发送到转换机（EXCHANGE），声明两个不同的队列（邮箱、手机），并绑定到交换机。这样生产者只需要发布一次消息，两个队列都会接收到消息发给对应的消费者。



手写代码 【白板编程】

1. 实现一个BST【二叉搜索树】

2. 实现一个HashMap【put/get方法】

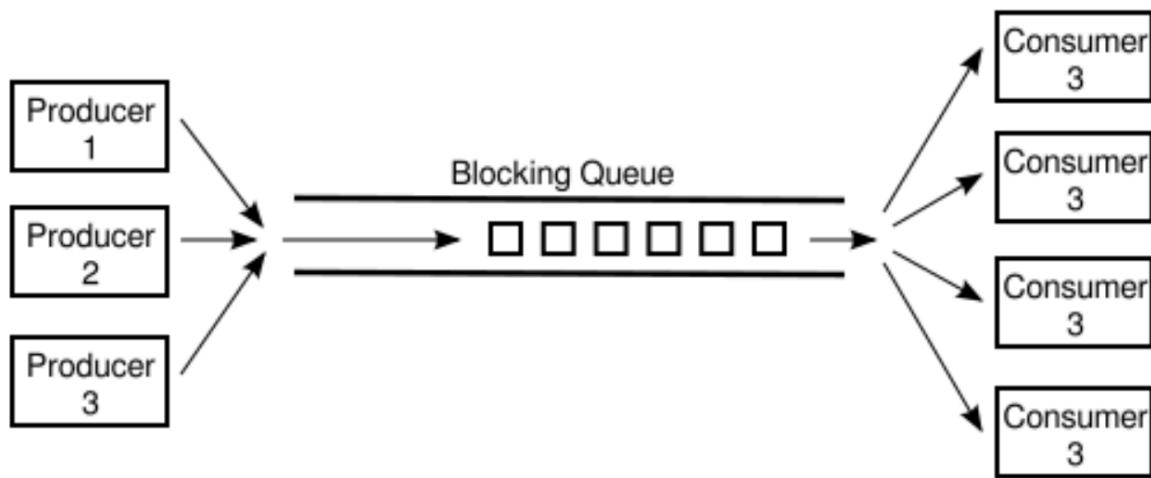
- 主要是设计Entry对象，包含K-V，以及指针next
- 需要继承MyMap<K,V>接口，注意这里泛型的使用

参考：<https://blog.csdn.net/feichitianxia/article/details/95808000>

3. 实现一个生产者消费者模型【使用wait和notify】

1. one生产者+one消费者

下面的blockingqueue可以使用LinkedList配合wait+notify使用



- 使用LinkedList实现缓冲区

```

package _00_Java_language;

import java.util.LinkedList;
import java.util.Random;

class Storage {
    private int maxSize;
    private LinkedList<Integer> storage;

    public Storage(){
        maxSize = 10;
        storage = new LinkedList<>();
    }

    //往缓冲区插入一个元素
    public synchronized void add(){
        while(storage.size() == maxSize){
            try{
                this.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        int random = new Random().nextInt(1000);
        storage.add(random);
        System.out.println("Set:" + random + ",current size :" +
storage.size());
        this.notifyAll();
    }

    //从缓冲区取出一个元素
    public synchronized void get(){
        while(storage.size() == 0){
    
```

```

        try{
            this.wait();
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }

    System.out.printf("Get: %d ,current size :%d \n",
storage.poll(),storage.size());
    this.notifyAll();
}

}

class Producer implements Runnable{
    private Storage storage;
    public Producer(Storage storage) {
        this.storage = storage;
    }

    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            storage.add();
        }
    }
}

class Consumer implements Runnable{
    private Storage storage;
    public Consumer(Storage storage) {
        this.storage = storage;
    }

    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            storage.get();
        }
    }
}

public class ProducerConsumer {
    public static void main(String[] args) {
        Storage storage = new Storage();

        Producer producer = new Producer(storage);
        Thread t1 = new Thread(producer);

        Consumer consumer = new Consumer(storage);
        Thread t2 = new Thread(consumer);
    }
}

```

```
        t2.start();
        t1.start();
    }
}
```

2. one生产者+mult消费者

3. 为什么要使用while进行循环判断

如果是多个消费者出现的环境，

一个消费者A唤醒了另一个消费者B，

线程B从上次wait的地方开始执行，然后如果这时候队列里面没有产品，那么B开始消费，就会产生数组越界。

参考：<https://blog.csdn.net/worldchinalee/article/details/83790790>

4. wait/notify为什么要配合synchronized使用

当一个线程在执行synchronized 的方法内部，调用了wait()后，该线程会释放该对象的锁，然后该线程会被添加到该对象的等待队列中（waiting queue），只要该线程在等待队列中，就会一直处于闲置状态，不会被调度执行。要注意wait()方法会强迫线程先进行释放锁操作，所以在调用wait()时，该线程必须已经获得锁，否则会抛出异常。由于wait()在synchronized的方法内部被执行，锁一定已经获得，就不会抛出异常了。

```
// 线程 A 的代码
synchronized(obj_A)
{
    while(!condition){
        obj_A.wait();
    }
    // do something
}
```

```
// 线程 B 的代码
synchronized(obj_A)
{
    if(!condition){
        // do something ...
        condition = true;
        obj_A.notify();
    }
}
```

详细解

答：<https://itimetraveler.github.io/2017/11/10/%E3%80%90Java%E3%80%91%E7%94%9F%E4%BA%A7%E8%80%85%E6%88%E8%B4%B9%E8%80%85%E6%A8%A1%E5%BC%8F%E7%9A%84%E5%AE%9E%E7%8E%BO/>

<https://my.oschina.net/u/2309504/blog/544086>

参考：<https://vinfai.iteye.com/blog/2082272>

4. 实现一个阻塞队列

1. ArrayBlockingQueue

ArrayBlockingQueue是数组实现的线程安全的有界的阻塞队列。线程安全是指，ArrayBlockingQueue内部通过“互斥锁”保护竞争资源，实现了多线程对竞争资源的互斥访问。而有界，则是指ArrayBlockingQueue对应的数组是有界限的。阻塞队列，是指多线程访问竞争资源时，当竞争资源已被某线程获取时，其它要获取该资源的线程需要阻塞等待；而且，ArrayBlockingQueue是按 FIFO（先进先出）原则对元素进行排序，元素都是从尾部插入到队列，从头部开始返回。

注意：ArrayBlockingQueue不同于ConcurrentLinkedQueue，ArrayBlockingQueue是数组实现的，并且是有界限的；而ConcurrentLinkedQueue是链表实现的，是无界限的。

2. 使用ReentrantLock实现阻塞队列

2.1 为什么要持有共同的ReentrantLock

保证每个时候只有一个线程在对addIndex和getIndex进行操作【自增操作不是原子性的】

2.2 为什么要使用Condition

- Condition中的await()方法相当于Object的wait()方法，Condition中的signal()方法相当于Object的notify()方法，Condition中的signalAll()相当于Object的notifyAll()方法。不同的是，Object中的这些方法是和同步锁捆绑使用的；而Condition是需要与互斥锁/共享锁捆绑使用的。
- Condition它更强大的地方在于：能够更加精细的控制多线程的休眠与唤醒。对于同一个锁，我们可以创建多个Condition，在不同的情况下使用不同的Condition。
 - 例如，假如多线程读/写同一个缓冲区：当向缓冲区中写入数据之后，唤醒“读线程”；当从缓冲区读出数据之后，唤醒“写线程”；并且当缓冲区满的时候，“写线程”需要等待；当缓冲区为空时，“读线程”需要等待。

```
package _00_Java_language._multi_thread;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ArrayBlockingQueue<T> {
    private Lock lock = new ReentrantLock();
```

```
private Object[] item;
private int addIndex, getIndex, count;
private Condition getCondition = lock.newCondition();
private Condition addCondition = lock.newCondition();

public ArrayBlockingQueue(int size){
    this.item = new Object[size];
}

public void add(T t){
    lock.lock();
    try{
        System.out.println("正在ADD对象 " + t);
        while(count == item.length){
            System.out.println("队列已满, 阻塞ADD线程");
            addCondition.await();
        }
        //队列未满, 添加元素, 计数器加1
        item[addIndex++] = t;
        count++;
        //如果ADD指针指向末尾, 那么重置
        if(addIndex == item.length) addIndex = 0;
        System.out.println("唤醒GET线程");
        getCondition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public T get(){
    lock.lock();
    T t = null;
    try{
        while(count == 0){
            System.out.println("队列空了, 阻塞GET线程");
            getCondition.await();
        }
        //队列没空
        t = (T) item[getIndex++];
        System.out.println("正在GET对象 " + t);
        count--;
        if(getIndex == item.length) getIndex = 0;
        System.out.println("唤醒ADD线程");
        addCondition.signal();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
```

```

        lock.unlock();
    }
    return t;
}

public static void main(String[] args) {
    final ArrayBlockingQueue queue = new ArrayBlockingQueue(3);
    new Thread(new Runnable() {
        @Override
        public void run() {
            for(int i=0; i < 3; i++)
                queue.add(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
    new Thread(new Runnable() {
        @Override
        public void run() {
            for(int i=0; i < 3; i++){
                queue.get();
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }).start();
}
}

```

5. topK问题 【有1亿个浮点数，如果找出其中最大的10000个？】

1. 单机条件

- 首先是去重
 - 【Hash】如果这1亿个数里面有很多重复的数，先通过Hash法，把这1亿个数字去重复，这样如果重复率很高的时候，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的10000个数。
- 然后是构建最小堆。

- 首先读入前10000个数来创建大小为10000的最小堆，建堆的时间复杂度为 $O(m\log m)$ (m 为数组的大小即为10000)，然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至1亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有10000个数字。该算法的时间复杂度为 $O(nm\log m)$ ，空间复杂度是10000（常数）。

2. 多机条件

(1) 单机+单核+足够大内存 如果需要查找10亿个查询次（每个占8B）中出现频率最高的10个，考虑到每个查询词占8B，则10亿个查询次所需的内存大约是 $10^9 * 8B = 8GB$ 内存。如果有这么大内存，直接在内存中对查询次进行排序，顺序遍历找出10个出现频率最大的即可。这种方法简单快速，使用。然后，也可以先用HashMap求出每个词出现的频率，然后求出频率最大的10个词。

(2) 单机+多核+足够大内存 这时可以直接在内存总使用Hash方法将数据划分成n个partition，每个partition交给一个线程处理，线程的处理逻辑同（1）类似，最后一个线程将结果归并。该方法存在一个瓶颈会明显影响效率，即数据倾斜。每个线程的处理速度可能不同，快的线程需要等待慢的线程，最终的处理速度取决于慢的线程。而针对此问题，解决的方法是，将数据划分成 $c > n$ 个partition ($c > 1$)，每个线程处理完当前partition后主动取下一个partition继续处理，知道所有数据处理完毕，最后由一个线程进行归并。

(3) 单机+单核+受限内存 这种情况下，需要将原数据文件切割成一个一个小文件，如次啊用 $\text{hash}(x)\%M$ ，将原文件中的数据切割成M小文件，如果小文件仍大于内存大小，继续采用Hash的方法对数据文件进行分割，知道每个小文件小于内存大小，这样每个文件可放到内存中处理。采用（1）的方法依次处理每个小文件。

(4) 多机+受限内存 这种情况，为了合理利用多台机器的资源，可将数据分发到多台机器上，每台机器采用（3）中的策略解决本地的数据。可采用hash+socket方法进行数据分发。

6. 实现一个LRU算法

1. 双向链表+HashMap

1. 为什么需要HashMap，HashMap用来存放什么

`HashMap<K, CacheNode> caches` HashMap的作用是双向链表查询操作的时间复杂度 $O(n)$ ，使用HashMap可以直接获取CacheNode

2. 双向链表的更新方案

- put操作
 - 如果容量没满，直接放在头部
 - 如果容量已满，那么删除尾部节点，然后放在头部
- get操作
 - 如果命中，拿出来放在头部
 - 如果没有命中...

3. Code

```
import java.util.HashMap;
import java.util.Map.Entry;
```

```
import java.util.Set;

public class LRUcache<K, V> { //类似于HashMap, 需要使用K-V泛型

    private int currentCacheSize;
    private int CacheCapcity;
    private HashMap<K,CacheNode> caches;
    private CacheNode first;
    private CacheNode last;

    public LRUcache(int size){
        currentCacheSize = 0;
        this.CacheCapcity = size;
        caches = new HashMap<K,CacheNode>(size);
    }

    public void put(K k,V v){
        CacheNode node = caches.get(k);
        if(node == null){
            if(caches.size() >= CacheCapcity){

                caches.remove(last.key);
                removeLast();
            }
            node = new CacheNode();
            node.key = k;
        }
        node.value = v;
        moveToFirst(node);
        caches.put(k, node);
    }

    public Object get(K k){
        CacheNode node = caches.get(k);
        if(node == null){
            return null;
        }
        moveToFirst(node);
        return node.value;
    }

    public Object remove(K k){
        CacheNode node = caches.get(k);
        if(node != null){
            if(node.pre != null){
                node.pre.next=node.next;
            }
            if(node.next != null){

```

```

        node.next.pre=node.pre;
    }
    if(node == first){
        first = node.next;
    }
    if(node == last){
        last = node.pre;
    }
}

return caches.remove(k);
}

public void clear(){
    first = null;
    last = null;
    caches.clear();
}
}

private void moveToFirst(CacheNode node){
    if(first == node){
        return;
    }
    if(node.next != null){
        node.next.pre = node.pre;
    }
    if(node.pre != null){
        node.pre.next = node.next;
    }
    if(node == last){
        last= last.pre;
    }
    if(first == null || last == null){
        first = last = node;
        return;
    }

    node.next=first;
    first.pre = node;
    first = node;
    first.pre=null;
}

}

private void removeLast(){
    if(last != null){
        last = last.pre;
    }
}

```

```

        if(last == null){
            first = null;
        }else{
            last.next = null;
        }
    }

@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    CacheNode node = first;
    while(node != null){
        sb.append(String.format("%s:%s ", node.key, node.value));
        node = node.next;
    }

    return sb.toString();
}

class CacheNode{
    CacheNode pre;
    CacheNode next;
    Object key;
    Object value;
    public CacheNode(){
    }
}

public static void main(String[] args) {
    LRUCache<Integer, String> lru = new LRUCache<Integer, String>(3);

    lru.put(1, "a");      // 1:a
    System.out.println(lru.toString());
    lru.put(2, "b");      // 2:b 1:a
    System.out.println(lru.toString());
    lru.put(3, "c");      // 3:c 2:b 1:a
    System.out.println(lru.toString());
    lru.put(4, "d");      // 4:d 3:c 2:b
    System.out.println(lru.toString());
    lru.put(1, "aa");     // 1:aa 4:d 3:c
    System.out.println(lru.toString());
    lru.put(2, "bb");     // 2:bb 1:aa 4:d
    System.out.println(lru.toString());
    lru.put(5, "e");      // 5:e 2:bb 1:aa
    System.out.println(lru.toString());
    lru.get(1);           // 1:aa 5:e 2:bb
    System.out.println(lru.toString());
}

```

```

        lru.remove(11);      // 1:aa 5:e 2:bb
        System.out.println(lru.toString());
        lru.remove(1);       //5:e 2:bb
        System.out.println(lru.toString());
        lru.put(1, "aaa");   //1:aaa 5:e 2:bb
        System.out.println(lru.toString());
    }

}

```

7. 实现一个LFU算法【TODO】

LRU是最近最少使用页面置换算法(Least Recently Used),也就是首先淘汰最长时间未被使用的页面! LFU是最近最不常用页面置换算法(Least Frequently Used),也就是淘汰一定时期内被访问次数最少的页!

比如,第二种方法的时期T为10分钟,如果每分钟进行一次调页,主存块为3,若所需页面走向为2 1 2 1 2 3 4
注意,当调页面4时会发生缺页中断 若按LRU算法,应换页面1(1页面最久未被使用)但按LFU算法应换页面3(十分钟内,页面3只使用了一次)

可见LRU关键是看页面最后一次被使用到发生调度的时间长短,而LFU关键是看一定时间段内页面被使用的频率!

8. 三个线程顺序打印

考察多线程类的实际应用

建立三个线程A、B、C, A线程打印10次字母A, B线程打印10次字母B,C线程打印10次字母C,但是要求三个线程同时运行,并且实现交替打印,即按照ABCABCABC的顺序打印。

选择使用ReentrantLock,维护一个共有变量,每次打印的时候使用一个Lock锁住,打印完成之后解锁。

```

package _00_Java_language;

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

public class _002_ThreeThreadPrint {
    private static Lock lock=new ReentrantLock();
    private static int state=0;//通过state的值来确定是哪个线程打印

    static class ThreadA extends Thread{
        @Override
        public void run(){

```

```
for (int i = 0; i <10 ; ) {
    try{
        lock.lock();
        while(state%3==0){ // 多线程并发，不能用if，必须用循环测试等待条件，避免虚假唤醒
            System.out.print("A");
            state++;
            i++;
        }
    }finally{
        lock.unlock();
    }
}

static class ThreadB extends Thread{
    @Override
    public void run(){
        for (int i = 0; i <10 ; ) {
            try{
                lock.lock();
                while(state%3==1){
                    System.out.print("B");
                    state++;
                    i++;
                }
            }finally{
                lock.unlock();
            }
        }
    }
}

static class ThreadC extends Thread{
    @Override
    public void run(){
        for (int i = 0; i <10 ; ) {
            try{
                lock.lock();
                while(state%3==2){
                    System.out.print("C");
                    state++;
                    i++;
                }
            }finally{
                lock.unlock();
            }
        }
    }
}
```

```
        }
    }

    public static void main(String[] args) {
        new ThreadA().start();
        new ThreadB().start();
        new ThreadC().start();
    }
}
```

9. 实现一个环形队列

1. Java中如何实现队列

海量数据处理

1. 10g文件，只有2g内存，怎么查找文件中指定的字符串出现位置

2.

3.

综合题目

1. 分布式系统中的流量控制

1.令牌桶

令牌桶算法最初来源于计算机网络。在网络传输数据时，为了防止网络拥塞，需限制流出网络的流量，使流量以比较均匀的速度向外发送。令牌桶算法就实现了这个功能，可控制发送到网络上数据的数目，并允许突发数据的发送。

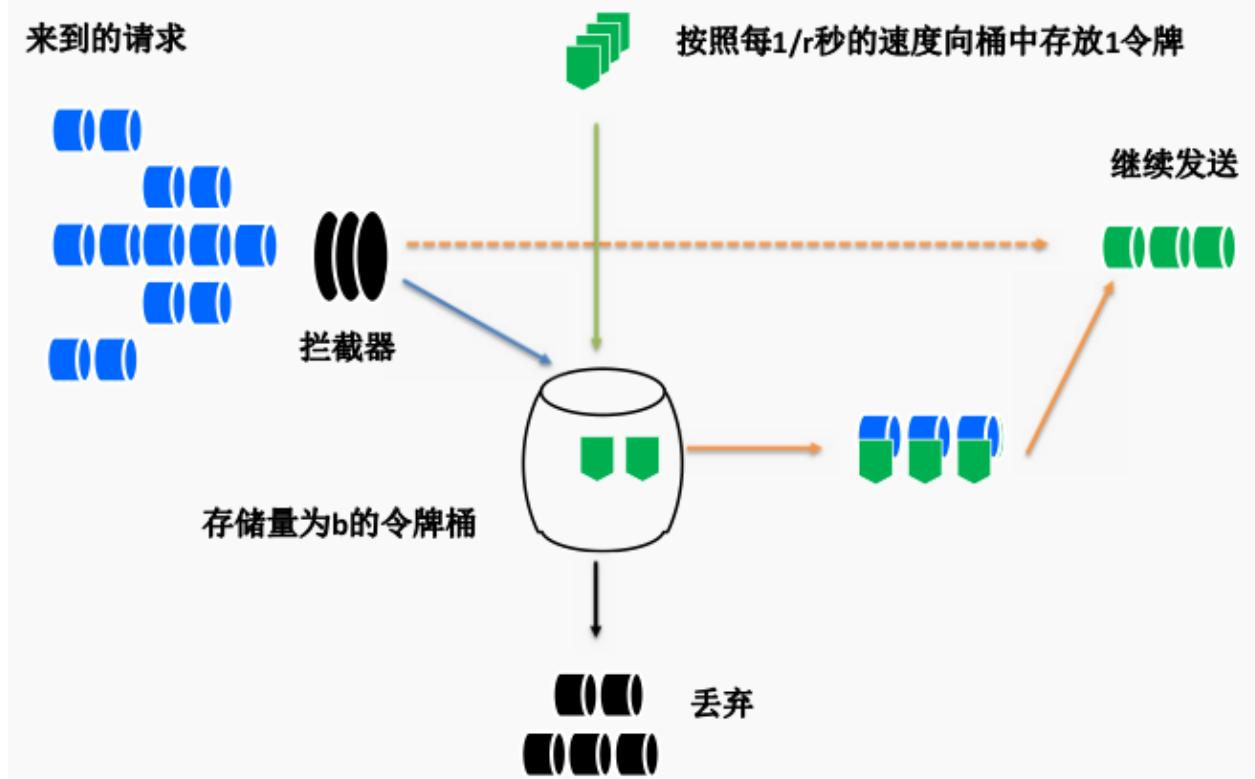
令牌桶算法是网络流量整形（Traffic Shaping）和速率限制（Rate Limiting）中最常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送。

大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。

传送到令牌桶的数据包需要消耗令牌。不同大小的数据包，消耗的令牌数量不一样。

令牌桶这种控制机制基于令牌桶中是否存在令牌来指示什么时候可以发送流量。令牌桶中的每一个令牌都代表一个字节。如果令牌桶中存在令牌，则允许发送流量；而如果令牌桶中不存在令牌，则不允许发送流量。因此，如果突发门限被合理地配置并且令牌桶中有足够的令牌，那么流量就可以以峰值速率发送。

令牌桶限流



- 假如用户配置的平均发送速率为 r ，则每隔 $1/r$ 秒一个令牌被加入到桶中（每秒会有 r 个令牌放入桶中）；
- 假设桶中最多可以存放 b 个令牌。如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃；
- 当一个 n 个字节的数据包到达时，就从令牌桶中删除 n 个令牌（不同大小的数据包，消耗的令牌数量不一样），并且数据包被发送到网络；
- 如果令牌桶中少于 n 个令牌，那么不会删除令牌，并且认为这个数据包在流量限制之外（ n 个字节，需要 n 个令牌。该数据包将被缓存或丢弃）；
- 算法允许最长 b 个字节的突发，但从长期运行结果看，数据包的速率被限制成常量 r 。对于在流量限制外的数据包可以以不同的方式处理：（1）它们可以被丢弃；（2）它们可以排放在队列中以便当令牌桶中累积了足够多的令牌时再传输；（3）它们可以继续发送，但需要做特殊标记，网络过载的时候将这些特殊标记的包丢弃。

2. 负载均衡算法

- Load Balance
 - 通过某种负载分担技术，将外部发送来的请求均匀分配到对称结构中的某一台服务器上，而接收到请求的服务器独立地回应客户的请求。

1. Round Robin 轮询

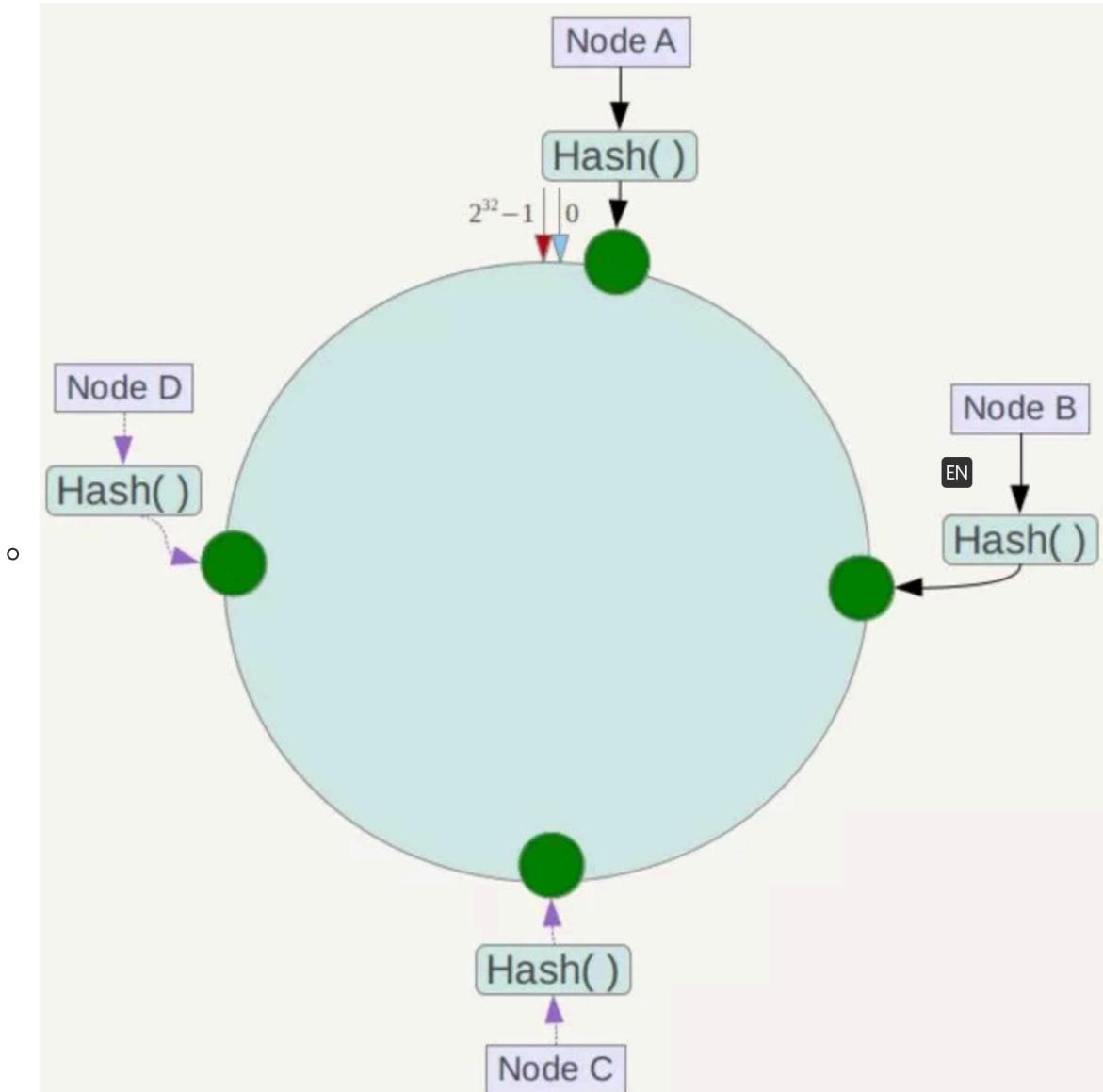
- 轮询调度 (Round Robin Scheduling) 算法就是以轮询的方式依次将请求调度不同的服务器，即每次调度执行 $i = (i + 1) \bmod n$ ，并选出第*i*台服务器。算法的优点是其简洁性，它无需记录当前所有连接的状态，所以它是一种无状态调度。
- 轮询调度算法假设所有服务器的处理性能都相同，不关心每台服务器的当前连接数和响应速度。当请求服务间隔时间变化比较大时，轮询调度算法容易导致服务器间的负载不平衡。

2. Weighted Round Robin 加权轮询

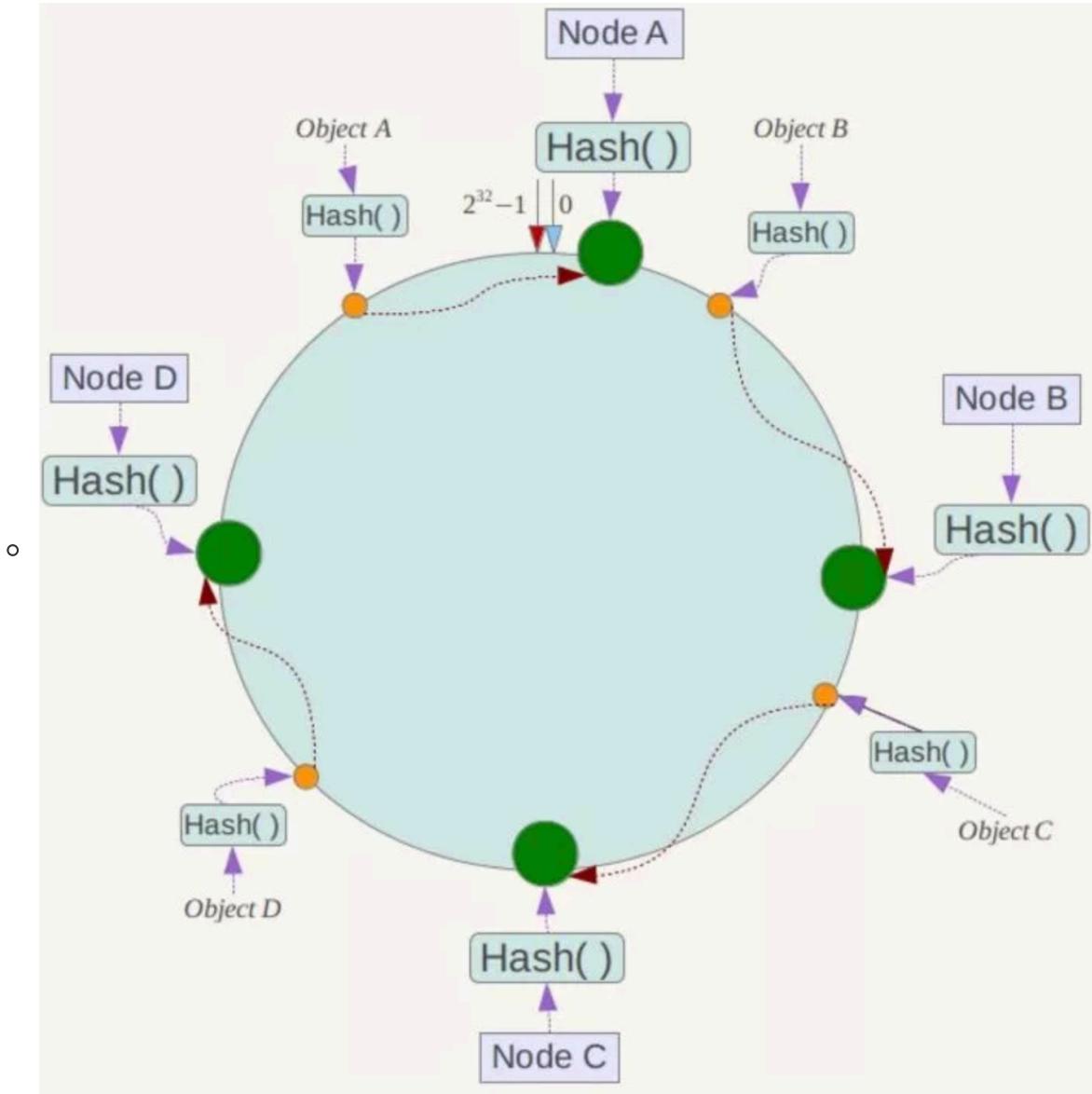
- 轮询算法并没有考虑每台服务器的处理能力，实际中可能并不是这种情况。由于每台服务器的配置、安装的业务应用等不同，其处理能力会不一样。所以，加权轮询算法的原理就是：根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。
- 加权轮询算法的结果，就是要生成一个服务器序列。每当有请求到来时，就依次从该序列中取出下一个服务器用于处理该请求。比如针对上面的例子，加权轮询算法会生成序列{c, c, b, c, a, b, c}。这样，每收到7个客户端的请求，会把其中的1个转发给后端a，把其中的2个转发给后端b，把其中的4个转发给后端c。收到的第8个请求，重新从该序列的头部开始轮询。

3. 一致性Hash算法

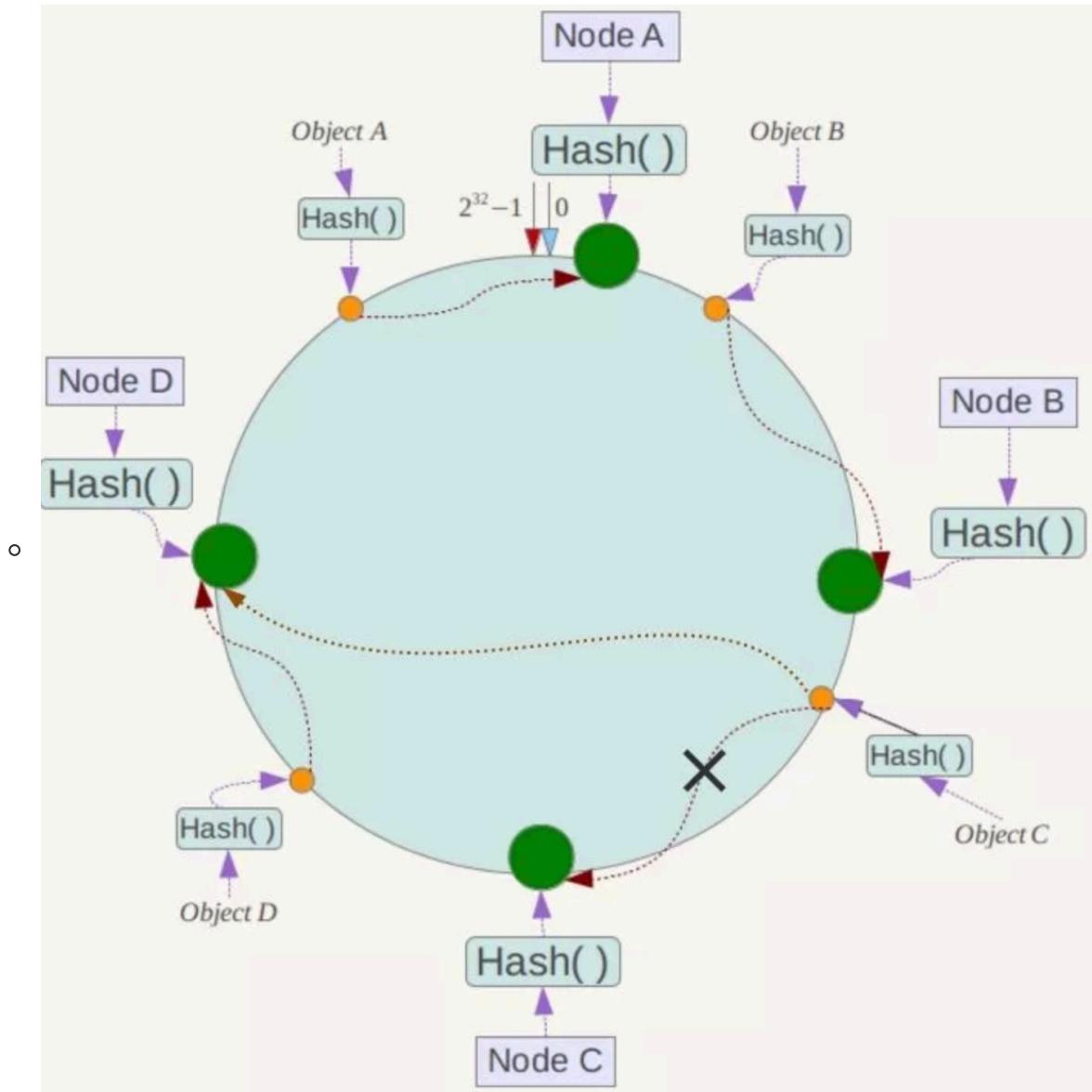
- 解决了什么问题
 - Redis集群中，直接使用Hash（取模算法），如果服务器数量变更，那么所有结果都会发生改变
- [1]首先具体可以选择服务器的IP或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用IP地址哈希后在环空间的位置如下：



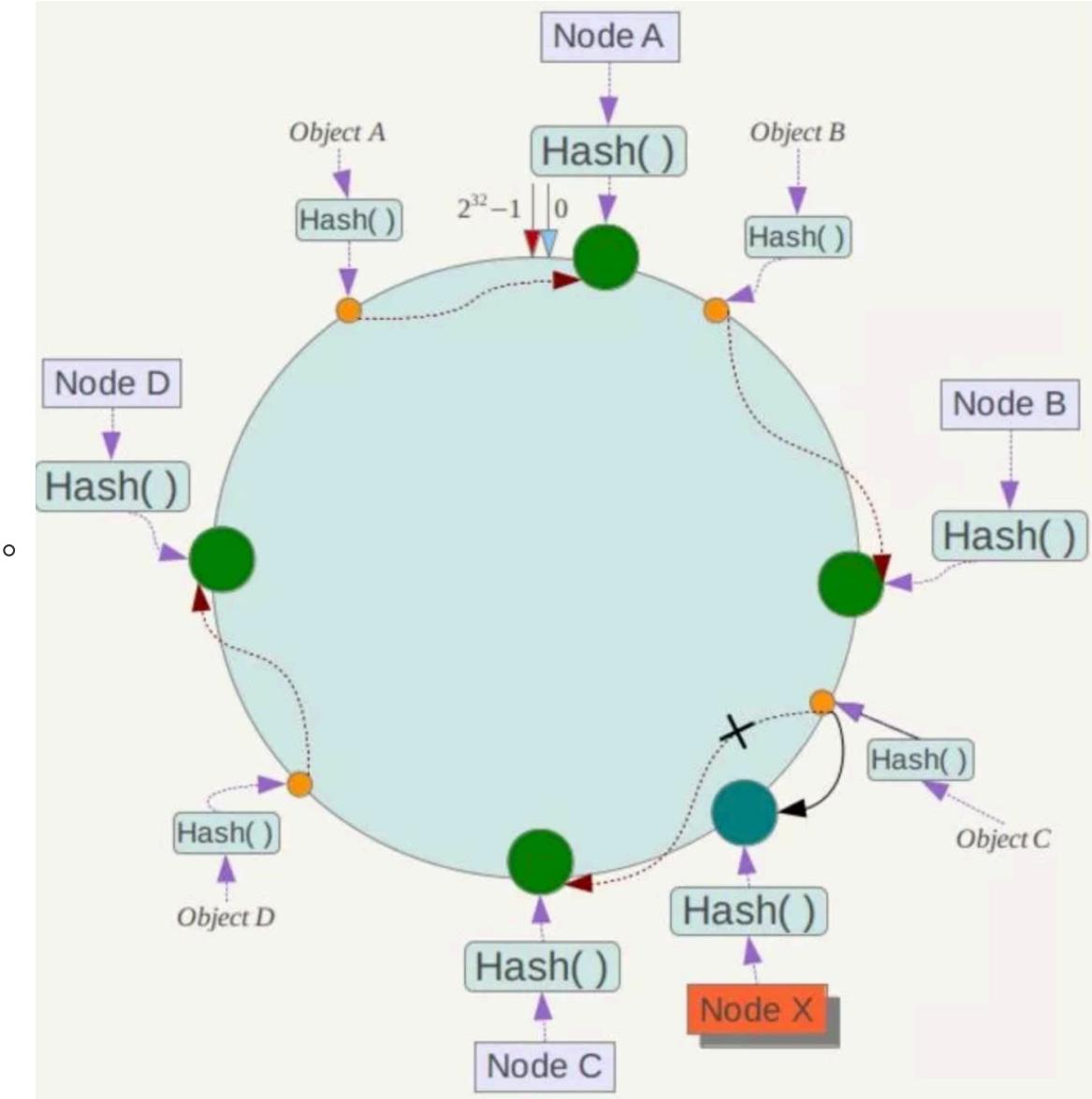
- [2]例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



- 如果删除了一个服务器节点
 - 现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性Hash算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响



- 如果增加了一个服务器节点
 - 如果在系统中增加一台服务器Node X



3. 面向对象六大原则

- **开闭原则：**对扩展开放,对修改关闭**，多使用抽象类和接口。
 - 比如代理模式，不修改原有的类，而是在基础上做扩展
- **里氏替换原则：**基类可以被子类替换，使用抽象类继承,不使用具体类继承。
 - 子类可以扩展父类的功能，但不能改变父类原有的功能。
- **依赖倒转原则：**要依赖于抽象,不要依赖于具体，针对接口编程,不针对实现编程。
- **接口隔离原则：**使用多个隔离的接口,比使用单个接口好，建立最小的接口。
- **迪米特法则：**一个软件实体应当尽可能少地与其他实体发生相互作用，通过中间类建立联系。
- **合成复用原则：**尽量使用合成/聚合,而不是使用继承。

0x8 实习总结

001 经验

1. 你在项目中遇到的最大的困难是什么？是如何解决的？

- 开发携程对应的接口，需求是给出针对携程服务CDN调度节点的信息[IP和运营商]
- 原本实现方案是直接查询跨部门API，组装JSON返回结果
- 解决问题1-给定内部接口响应太慢，使用Redis作为缓存，缓存调度信息
- 解决问题2-收到携程邮件报警信息，节点调度信息出错，Redis缓存时间设置为7天，携程访问频率为30min/，所以出现调度信息更新不及时的问题
- 尝试解决方案
 - 新增内部访问API，找一台服务器，定时访问服务（频率≈15min/），模拟携程访问，更新Redis，这种问题是OpenCDN-API后台设置了HTTP访问请求超时时间，容易更新失败
 - 直接在Centos开发机上通过crontab部署定时任务，单独配置PHP访问跨部门API，刷新Redis
- 总结
 - 考虑问题要全面，对于要上线的接口需要充分测试。
 - 重构代码的情况比较正常，开发要面向需求。

2. 实习收获

- 工程能力
 - 学习一门新语言的能力：PHP
 - 对于新领域的学习姿势：CDN
- 软技能
 - 面向需求开发，需要和mentor以及客户进行交流，弄清楚XXAPI开发的作用、以及需求。
 - 有些时候需要跨部门合作，所以需要主动和不同部门的同事进行沟通。

002 相关知识

1. Nginx&PHP

1. Nginx和PHP之间怎样通信

Nginx和PHP之间有两种通信方式：Unix Socket和TCP Socket

(1)nginx.conf中配置php-fpm的pid

```

location ~ \.php$ {
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;;
    fastcgi_pass unix:/var/run/php5-fpm.sock;
    fastcgi_index index.php;
}

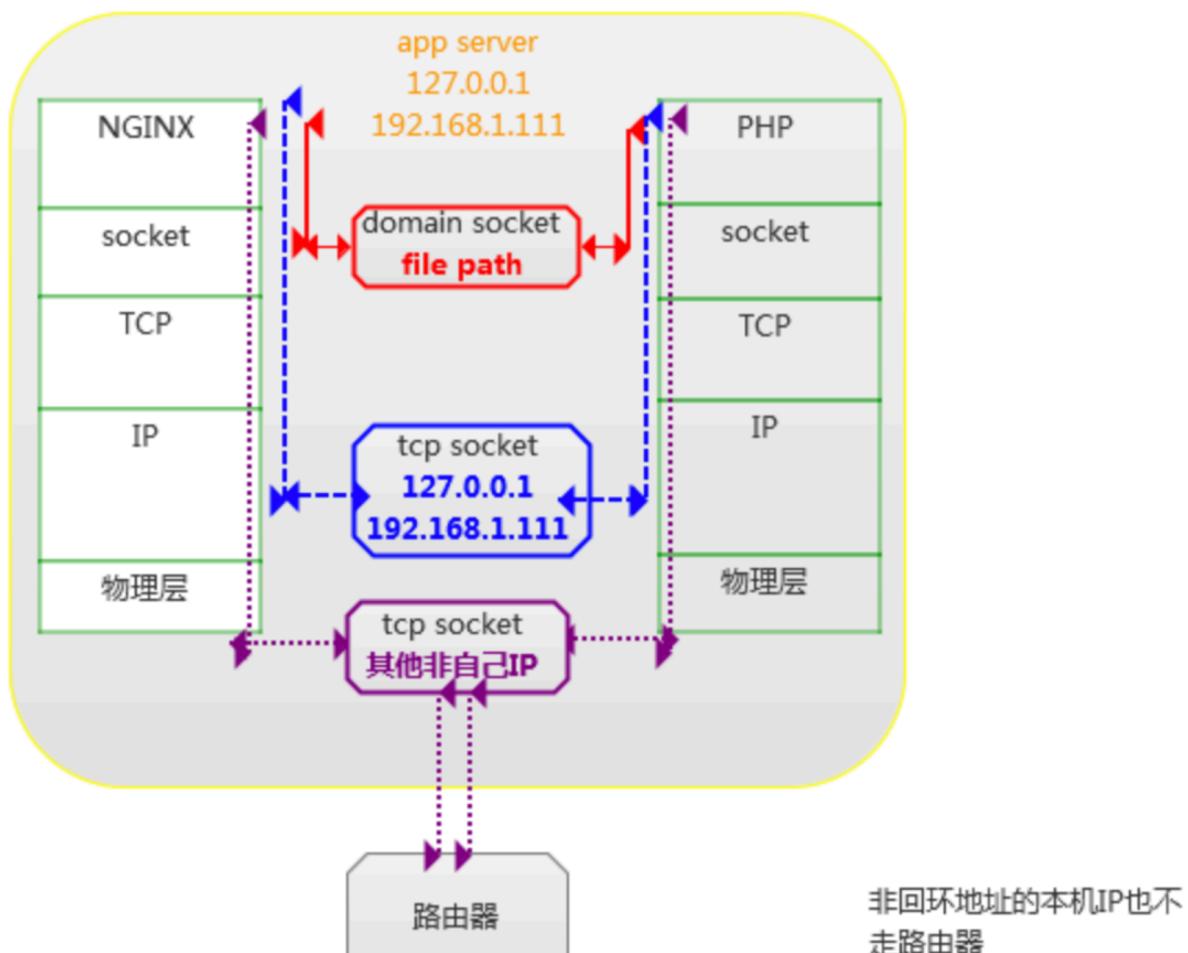
```

(2)nginx.conf中配置php-fpm的IP+端口

```

location ~ \.php$ {
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;;
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
}

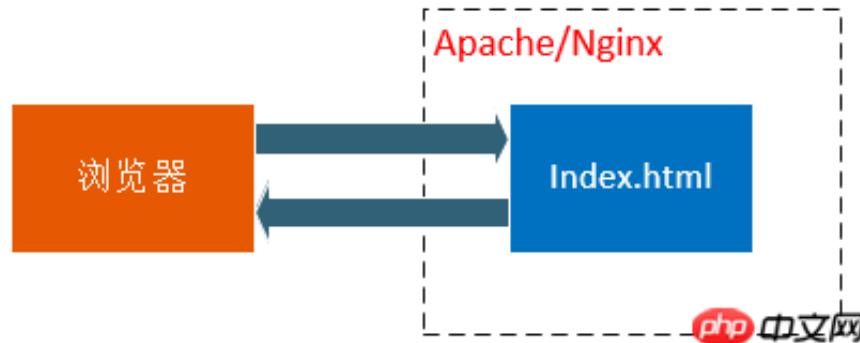
```



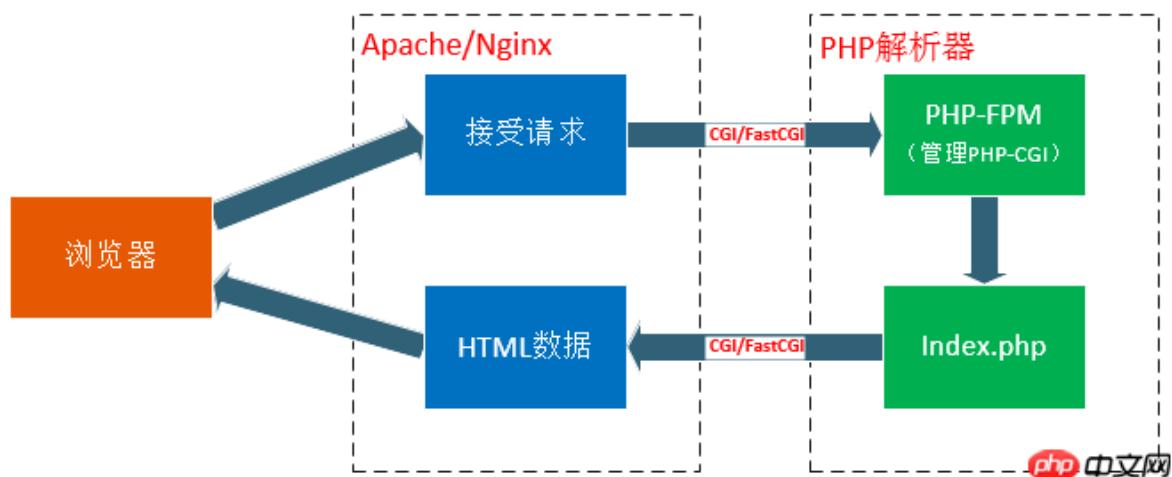
- unix socket减少了不必要的tcp开销，而tcp需要经过loopback，还要申请临时端口和tcp相关资源。但是，unix socket高并发时候不稳定，连接数爆发时，会产生大量的长时缓存，在没有面向连接协议的支撑下，大数据包可能会直接出错不返回异常。tcp这样的面向连接的协议，多少可以保证通信的正确性和完整性。
- 选择建议：如果是在同一台服务器上运行的nginx和php-fpm，并发量不超过1000，选择unix socket，因为是本地，可以避免一些检查操作(路由等)，因此更快，更轻。如果面临高并发业务，我会选择使用更可靠的tcp socket，以负载均衡、内核优化等运维手段维持效率。

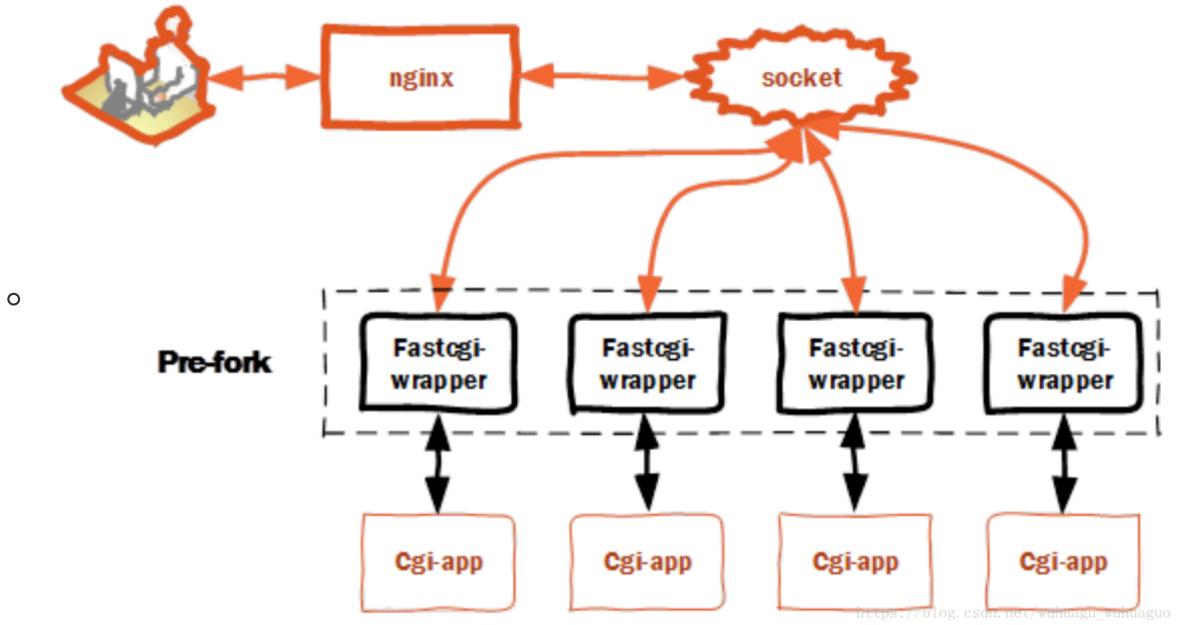
2. 处理请求的过程

- 请求index.html: 在整个网站架构中, Web Server (如Apache) 只是内容的分发者。举个栗子, 如果客户端请求的是 index.html, 那么Web Server会去文件系统中找到这个文件, 发送给浏览器, 这里分发的是静态数据。



- 请求index.php: 当Web Server收到 index.php 这个请求后, 会启动对应的 CGI 程序, 这里就是 PHP的解析器。接下来PHP解析器会解析php.ini文件, 初始化执行环境, 然后处理请求, 再以规定CGI规定的格式返回处理后的结果, 退出进程, Web server再把结果返回给浏览器。这就是一个完整的动态PHP Web访问流程, 接下来再引出这些概念, 就好理解多了,





关于基本概念的理解

- CGI: 是 Web Server 与 Web Application 之间数据交换的一种协议。
- FastCGI: 同 CGI, 是一种接口/协议, 但比 CGI 在效率上做了一些优化。同样, SCGI 协议与 FastCGI 类似。
- PHP-CGI: 是 PHP (Web Application) 对 Web Server 提供的 CGI 协议的接口程序。
- PHP-FPM(PHP-FastCGI Process Manager): 是 PHP (Web Application) 对 Web Server 提供的 FastCGI 协议的接口程序, 额外还提供了相对智能一些任务管理。
- 当访问index.php时, 具体发生了什么?

1. 根据nginx.conf配置文件, 找到对应的server, 然后是对应的location 【路径复合正则匹配规则】

```

location ~ \.php$ {
    root /home/admin/web/nginx/html/;
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME
    /home/admin/web/nginx/html/$fastcgi_script_name;
    include fastcgi_params;
}

```

2. 将请求通过127.0.0.1:9000 (IP+端口) 分发给FastCGI进程处理
3. FastCGI进程管理器php-fpm自身初始化, 启动主进程php-fpm和启动start_servers个fastcgi子进程。主进程php-fpm主要是管理fastcgi子进程, 监听9000端口, fastcgi子进程等待请求。子进程接受请求并且处理【返回响应&打印日志】

3. 什么是FastCGI? 什么是PHP-FPM

0x Linux

001 Linux常用命令

awk

简单来说awk就是把文件逐行的读入，以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

```
awk '{pattern + action}' {filenames}
```

1. 显示最近登录的5个用户名

```
[root@www ~]# last -n 5 <==仅取出前五行
root      pts/1    192.168.1.100  Tue Feb 10 11:21    still logged in
root      pts/1    192.168.1.100  Tue Feb 10 00:46 - 02:28  (01:41)
root      pts/1    192.168.1.100  Mon Feb  9 11:41 - 18:30  (06:48)
dmtsai   pts/1    192.168.1.100  Mon Feb  9 11:41 - 11:41  (00:00)
root      ttys1                               Fri Sep  5 14:09 - 14:10  (00:01)
```

如果只是显示最近登录的5个帐号

```
#last -n 5 | awk '{print $1}'
root
root
root
root
dmtsai
root
```

BEGIN 和 END 可以确定开始执行的命令和结束执行的命令

```
awk 'BEGIN{print "aaa"} {print $1} file.txt'
```

```
awk '{print $1} END{print "bbb"} file.txt'
```

grep

```
ps -ef | grep php
```

```
history | grep root
```

1. 查询文件/文件夹下特定的字符串

```
grep 'string' fileName
```

```
grep 'string' dirPath/*
```

2. 查询进程

```
--
```

sed

sed命令是一个很强大的文本编辑器，可以对来自文件、以及标准输入的文本进行编辑。

执行时，**sed**会从文件或者标准输入中读取一行，将其复制到缓冲区，对文本编辑完成之后，读取下一行直到所有的文本行都编辑完毕。

所以**sed**命令处理时只会改变缓冲区中文本的副本，如果想要直接编辑原文件，可以使用-i选项或者将结果重定向到新的文件中。

```
sed [options] commands [inputfile...]
```

1. 输出文本的1~5行

```
sed -n '1,5 p' test.txt
```

- -p: print打印
- -n: 取消默认输出

vim

1. 查询某个字符串出现的次数

```
:%s/字符串/&/gn
```

- df
 - 显示磁盘使用情况
- top
 - top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。

curl

mkdir

cd

sudo

su

ls

```
ls -l
```

ps

要对进程进行监测和控制,首先必须要了解当前进程的情况,也就是需要查看当前进程,ps命令就是最基本进程查看命令。使用该命令可以确定有哪些进程正在运行和运行的状态、进程是否结束、进程有没有僵尸、哪些进程占用了过多的资源等等.总之大部分信息都是可以通过执行该命令得到。ps是显示瞬间进程的状态，并不动态连续；如果想对进程进行实时监控应该用top命令。

top

top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。

top显示系统当前的进程和其他状况,是一个动态显示过程,即可以通过用户按键来不断刷新当前状态.如果在前台执行该命令,它将独占前台,直到用户终止该程序为止. 比较准确的说,top命令提供了实时的对系统处理器的状态监视.它将显示系统中CPU最“敏感”的任务列表.该命令可以按CPU使用.内存使用和执行时间对任务进行排序。

Shell基本语法

1. 接受传入参数

```
#!/bin/bash
echo "执行的文件名: $0"; //第一个参数是文件名
echo "第一个参数为: $1";
echo "第二个参数为: $2";
echo "第三个参数为: $3";
```

2. 变量初始化&赋值&打印

```
a="123";
echo $a;
```

3. 判断条件

```
#!/bin/bash
str1=liushen;
str2=liuting;
if [ $str1 = $str2 ]
then
    echo equal;
    echo equal2;
else
    echo not equal;
    echo not equal2;
fi
```

4. 循环处理

5. 字符串拼接

6. 执行字符串形式的命令

0x9 方法迭代

- 『要获得什么，就需要牺牲什么』
 - e.g. 比如TCP协议和UDP协议的对比学习：TCP协议能够保证传输数据的完整、有序，UDP协议是不可靠的，不需要接收方确认。但是，从另一方面来说，UDP协议的效率更高。
 - e.g. 整个学习过程也是如此，要熟悉算法题，就需要付出足够多的时间&精力刷题总结。
 - e.g. 所有的操作都是有代价的，有得有失，完美的解决方案往往难以奢求。
- 『新技术的起源是首先关注的』
 - 相对于old things，新技术的出现是为了解决以前技术的问题
 - e.g. Spring框架的出现是为了让程序员更加专注于业务逻辑，摒弃原生技术对于代码形式上的约束。
 - e.g. Redis数据库在MySQL数据库之后出现，是为了解决解决高并发场景下的性能问题。
 - e.g. JDK1.8的更新，提出了函数式编程，lambda表达式是为了让代码更加clean，更加提现业务逻辑。
- 『从实际问题出发，搭建知识系统』
 - 对于每一种技术或者体系，往往会有-套主要的解决问题的体系，但是往往需要耗费大量的时间去实践。
 - 从实际问题出发，"面经"表示面试官比较感兴趣的知识点：
 - 说明是实际项目中比较需要的point
 - 说明是能够提现被面试者专业知识积累的point
 - 从实际问题出发，比如使用"RabbitMQ"，就要考虑到在分布式系统中消息队列经常需要面对的问题：宕机如何处理，如何实现数据的持久化
- 『脱离形式主义的大坑』
 - 保证笔记的良好风格，方便复习以及形成知识体系
 - 但是笔记最终的目的是为了提高效率，不是为了复制整理，形式不是一切
 - 一级标题：核心内容，二级标题&三级标题：主题，四级标题：常见应用和细节
- 『What matters most leading first』
 - 面经常考 > 面经出现 > 知识系统细节
- 『面试的引导性』
 - 你了解Java中的集合框架吗
 - 你谈到了ArrayList和LinkedList，那么有线程安全的List实现类吗
 - 既然提到了线程安全，你知道Java里的锁吗

- 既然提到了Java里的锁，那你知道乐观锁吗
 - 既然提到了乐观锁，那你知道ABA问题吗
 - 既然提到了版本控制机制，那你知道MySQL MVCC吗
 - 既然提到了MySQL MVCC，那你知道MySQL隔离级别吗
- 你谈到了HashMap，那你说说HashMap的实现原理
 - 你谈到了红黑树，那你说说红黑树的定义
 - 红黑树是AVL树的改进，你知道其他改进吗
 - 你谈到了B+树，你说说B+树的应用吗
 - 你谈到了索引，那你说说为什么不使用B树吗
 - 你谈到了B树，能说说操作系统的文件系统为什么要用它吗
 - 我们谈到操作系统，那你说说内存管理吧
 - ...
 - 你知道HashMap的扩容机制吗
 - 既然谈到了扩容，你知道缩容机制吗

0x10 TODO List

后端开发常问面试题集锦（附链接）：<https://blog.csdn.net/andong154564667/article/details/80117546>

2020秋招面经大汇总！（岗位划分）：<https://www.nowcoder.com/discuss/205497>

面试题目参考：<https://blog.csdn.net/xlgen157387/article/details/88051362>

博客项目吸收：https://troywu0.gitbooks.io/spark/content/javaduo_xian_cheng.html

https://www.sohu.com/a/256461492_129720

知识储备：<https://www.nowcoder.com/discuss/216588>

```
# 结尾，关于8月的一些长远目标
1. 整理LeetCode算法题目
2. 整体知识系统：计算机网络+数据库+JAVA+操作系统
3. 针对面试需要考虑的细节，进行打磨
```

1. 看过哪些书

1. Java

《深入理解JVM虚拟机》

《Java8实战》

《Effective Java》

2. 计算机网络

《图解HTTP》

《计算机网络—自顶向下方法》

3. 其他

《Redis入门》

《分布式消息中间件实践》

《Spring+MyBatis企业应用实战》

《重构》

《C和指针》

《浪潮之巅》

0x11 the end

1. 关于面试的一些经验

1. 对于某些知识，就算忘掉了细节，但是也要说出自己知道的部分，答的不全总比不回答好【不回答会让面试官觉得这个人基础不行，学习的广度不能满足要求】

2. 面试进入后期，现在的问题是解决『核心问题』，那么哪些是核心问题？

手写一个LRU算法

AOP底层实现原理

操作系统内存模型

计算机网络典型协议

掌握常见的几种设计模式

