1 预备知识

如果只是给一个普通项目提供搜索支持，你不需要多少预备知识，可以跳过本节。如果你想把搜索作好，建议你看本节。本节假定读者已经具备计算机专业本科水平。

本预备知识包含了搜索引擎一般需要使用的而普通课本上面又没有详细描述的数据结构和压缩算法。

1.1 数据结构

1.1.1 Hash 函数

定义：Hash 函数把任意范围的 key 映射到一个相对较小的整数范围内。比如，md5 hash 算法就是把任意字节数组映射到 128 位的整数上。

Hash 函数在很多领域都有应用，比如，模式匹配、密码学。它们都额外的定义了 Hash 函数的其他特点，如密码学要求 Hash 函数接近于单向函数。我们这里只讨论 Hash 函数在 Hash table 结构里面的应用。

1.1.1.1 常用 Hash 函数介绍

Hash 函数一般会分为 2 种，即 Hash table 用的 hash 和密码学上面应用的 Hash。这里分开介绍。

1.1.1.1.1 Hash table 的 Hash 函数

应用在 Hash table 上的 Hash 函数的一个主要特点是输出一般为 32 或者 64 位。常见的算法有很多种，有关这些 Hash 算法的实现代码，可以参考笔者博客上的文章：Hash 算法实现。

1.1.1.1.2 密码学上的 Hash 函数

密码学上的 Hash 函数第一重要的是其伪单向性。故一般其计算都特别复杂，结果位数也很长。我们常见的 Hash 函数有如下几个：

- Adler-32：定义在 rfc1950 里面，是一种乘法 Hash 算法，结果是 32 位整数；
- CRC32：循环冗余码，结果是 32 位整数；
- MD 系列：md3, md4, md5 等，结果都是 128 位整数；考虑到 md5 冲突的易构造性，建议不要用它做重要消息签名；

- RipedMd 系列：有 128/160/256/320 位 4 种规范，但 RipedMd-256/320 没有被认真设计，其安全度和 128/160 这 2 种规范一样；
- SHA 系列：有 160/256/384/512 位 4 种规范；
- Tiger：192 位，专门为 64 位机器优化；
- WhirlPool：512 位。

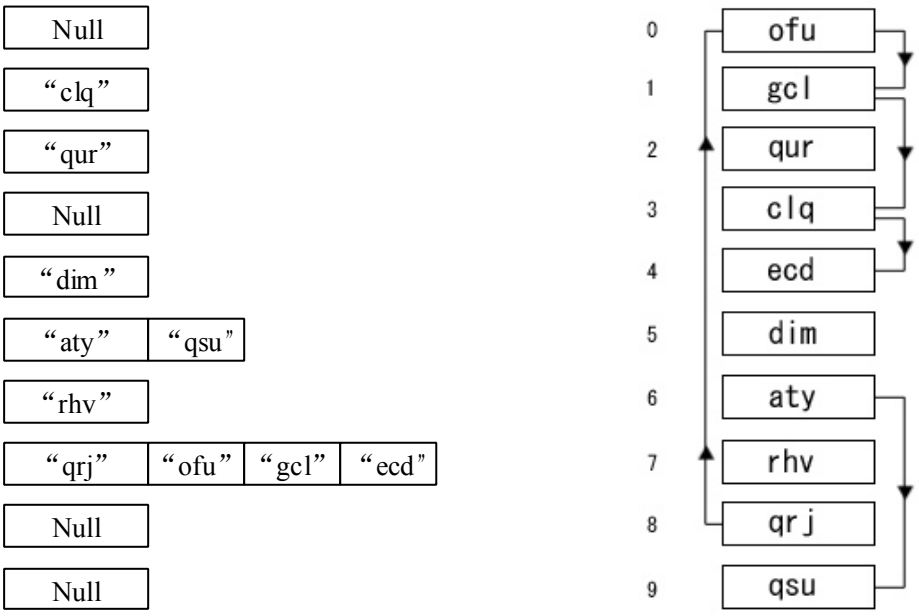
这里，需要注意的是，美国的国家标准推荐使用如下几个 Hash 算法：RipedMd 系列，SHA 系列，Tiger 和 WhirlPool。具体的说明大家可以去看密码学相关书籍（推荐《密码学基础（英文版）》和《现代密码学理论与研究》）或者学习著名的开源库 Crypto++或者参考笔者的博客：Hash 散列算法详细解析。

1.1.1.2Hash table 处理碰撞的技术

- Open addressing（closed hashing）：探测方式。通过选择不同的位置来存放数据。
  - 线形探测：计算公式为 next_pos = (cur_pos + m) mod N；其中 m 为常数，N 为 Hash table 数组长度；
  - 平方探测：常用计算公式为 next_pos = (cur_pos + cur_pos^2) mod N；
  - 二次 Hash：常用计算公式为 next_pos = (cur_pos + Hash2(key)) mod N；Hash2 表示第二个 Hash 函数。
- Chaining：链表方式。即把冲突的 key 串在一起。

以上两种碰撞处理技术很多数据结构书籍都详细描述过，本处不在赘述。下面看看其他的方法：

- Coalesced hashing：结合了 Open addressing 和 Chaining 方法，把 Chaining 的链表融合到 Hash table 的数组里面，从而减少了空间使用。下面用一个例子说明这种方式。输入 key 序列为{“qur”，“qrj”，“ofu”，“gcl”，“clq”，“ecd”，“aty”，“rhv”，“qus”}，左边图是基本 Chaining 方式的结果，右边图是本方式的结果：



Coalesced hashing 方式的处理方法为：利用线形探测找到一个空位，填入 key，然后把这个空位链接到相同映射值组成的链表上面。需要注意的是，Coalesced hashing 对定长的 key 才真正有用。

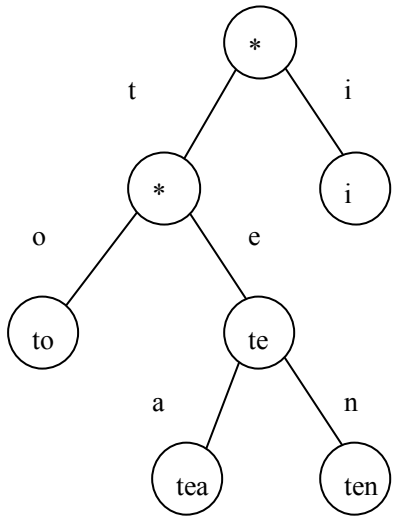
- Cuckoo hashing：基本思想是使用 2 个 hash 函数来处理碰撞，从而每个 key 都可以对应到 2 个位置。其具体的处理方式为：
  - 如果 key 对应的 2 个位置中有一个为空，key 就可以插入到那个位置；
  - 否则，任选一个位置，key 插入，把已经在那个位置的 key 踢出来；
  - 被踢出来的 key 需要重新插入；
  - 直到没有 key 被踢出为止。

  显然，这种方式是可能产生无限循环的，当出现无限循环的时候，就需要重新选择 hash 函数。

  Cuckoo hashing 有 2 个主要的变形：1）增加 hash 函数的个数；2）每个位置可以放 2 个 key。这 2 个的目的都是为了增加 Hash table 数组的利用率，因为基本的 Cuckoo hashing 只有 49%的利用率，而 3 个 Hash table 可以利用 91%的空间，每个位置保存 2 个 key 可以使利用率达到 80%。有人分析 Cuckoo hashing 比 Chaining 的方式更好。基本 Cuckoo Hashing 的实现，参考笔者博客文章：Cuckoo Hash 介绍。

1.1.2 Trie 树

也叫前缀数组，是一种 key 为字符串的联合数组。下图是一个简单示例：



Trie 树的一个主要特点是查询字符串 Key 很快。有关 Trie 树的介绍可以参考一般的数据结构书籍。基本的 Trie 树占用的内存很大，人们发明了基于数组的 Trie 结构。双数组 Trie 树其实是把 Trie 树中的所有节点拼接在一起，减少占用的空间。以上图来说，这棵树可以用 7 个（7 个节点，从上往下编号）数组表示：

- Node1[0:256]={null} ， Node1[‘t’]=Node2 ， Node1[‘i’]=Node3 ， isNodeHasData=false
- Node2[0:256]={null} ， Node2[‘o’]=Node4 ， Node2[‘e’]=Node5 ， isNodeHasData=false
- Node3[0:256]={null}，isNodeHasData=true

- Node4[0:256]={null}，isNodeHasData=true
- Node5[0:256]={null}，Node5[‘a’]=Node6，Node5[‘n’]=Node7，isNodeHasData=true
- Node6[0:256]={null}，isNodeHasData=true
- Node7[0:256]={null}，isNodeHasData=true

注意到共有 7 个 256 个元素的数组，数组元素表示对应节点的孩子节点对应的数组，数组里面大部分都是 Null 值，所以如果把它们组合在一起，可以大大减少空间占用。

在上面的这个例子中，可以简单的把所有数组混合起来，因为它们没有冲突；也可以把所有数组并排；前者组合后的数组大小为 256，后者组合后的数组大小为 256*7。实际中，直接混合数组会有很多冲突，而并排所有数组又十分浪费空间，故必须找一个好的办法来避免冲突同时减少空间浪费。文章“Fast and compact updating algorithms of a double-array structure”提出了一种基于空余空间链接的混合数组的算法，这是一种贪婪算法，能够较好地减少空间浪费。在实际应用中，我们发现这种算法的构建速度极慢，主要慢在贪婪算法上面。为了加快速度，我们在贪婪算法的基础上，跳过那些可能不匹配的位置，在多浪费一点空间的情况下，能够快速的构建双数组 Trie 树。

下面是双数组 Trie 树使用 java 实现的一个版本：

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.TreeMap;
import java.util.Map.Entry;

import lotusroots.algorithms.common.array.SimpleIntArrayList;
import lotusroots.algorithms.common.array.SimpleObjectArrayList;

/**
 * Trie 结构支持下的 Map<br>
 * 主要好处：<br>
 * 1，内存占用很少；<br>
 * 2，查询速度几乎是最快，好于其他任何 Map 结构；<br>
 * 3，删除很快，比 TreeMap 好；<br>
 * 4，加入速度可以，比 TreeMap 差；<br>
 * 5，修改飞快，比 TreeMap 好；<br>
 * <br>
 * 限制：<br>
 * 只能使用 String 作为 Key<br>
 *
 * @author Goodzzp 2007-5-10
 * @lastEdit Goodzzp 2007-5-10
*/
public class TrieMap<T>
{
	/**
	 * 状态列表
	 */
	SimpleIntArrayList m_states;
	/**
	 * 状态检测列表
	 */
	SimpleIntArrayList m_checks;
	/**
	 * 对应的对象
	 */
	SimpleObjectArrayList m_refObjs;

	/**
	 * 构造函数
	 * @param ds 字符串列表
	 * @param os 对应的对象列表
	 */
	public TrieMap(List<String> ds, List<T> os)
	{
		initial(ds, os);
	}
	/**
	 * 构造函数
	 */
	public TrieMap()
```

```
{
        List<String> strs = new ArrayList<String>();
        List<T> objs = new ArrayList<T>();
        strs.add("goodzzp provider it!");
        objs.add(null);
        initial(strs, objs);
}
/**
 * 初始化
 * @param ds 字符串列表
 * @param os 对象列表
 */
private void initial(List<String> ds, List<T> os)
{
        List<String> strs = new ArrayList<String>();
        List<Object> objs = new ArrayList<Object>();
        // 1, 排序
        TreeMap<String, Object> sortTree = new TreeMap<String, Object>();
        for(int i=0;i<ds.size();i++)
                sortTree.put(ds.get(i), os.get(i));
        Iterator<Entry<String, Object>> iter = sortTree.entrySet().iterator();
        while(iter.hasNext())
        {
                Entry<String, Object> e = iter.next();
                strs.add(e.getKey());
                objs.add(e.getValue());
        }
        // 2, 构建 trie 数组
        SimpleIntArrayList states = new SimpleIntArrayList();
        SimpleIntArrayList checks = new SimpleIntArrayList();
        SimpleObjectArrayList refObjs = new SimpleObjectArrayList();

        states.set(0, -1);      // 占位前 2 个状态
        states.set(1, 1);       // 初始以 1 开始
        m_scan = 1;
        create(strs,objs,states,checks,refObjs,0,0,strs.size(),1);

        m_states = states;
        m_checks = checks;
        m_refObjs = refObjs;
}

/**
 * 得到 key 对应的对象
 */
public T get(String key)
{
        int base = m_states.getQuick(1);
        int pos = 0;
        int lastPos = 1;
        for(int i=0;i<key.length();i++)
        {
                char c = key.charAt(i);
                pos = base + c;
                if(pos >= m_checks.size() || m_checks.getQuick(pos) != lastPos)
                        return null;
                base = m_states.getQuick(pos);
                lastPos = pos;
        }
        if(pos >= m_refObjs.size())
                        return null;
        else
                return (T)m_refObjs.getQuick(pos);
}
/**
 * 匹配 key 的最大前缀
 */
public T getByPartialMatch(String key)
{
        Object ret = null;
        int base = m_states.getQuick(1);
        int pos = 0;
        int lastPos = 1;
        for(int i=0;i<key.length();i++)
        {
                char c = key.charAt(i);
                pos = base + c;
                if(pos >= m_checks.size() || m_checks.getQuick(pos) != lastPos)
                        break;
                Object temp = m_refObjs.getQuick(pos);
                if(temp != null)
                        ret = temp;
                base = m_states.getQuick(pos);
                lastPos = pos;
        }
        return (T)ret;
}
```

```
	/**
	 * 保存的当前扫描到的位置
	 */
	int m_scan;

	/**
	 * 建立 Trie 结构
	 * @param strs 字符串列表
	 * @param objs 对象列表
	 * @param states 状态列表
	 * @param checks 状态检查列表
	 * @param refObjs 对应的对象
	 * @param depth 当前深度（第几个字符）
	 * @param first 字符串列表的开始位置
	 * @param last 结束位置
	 * @param lastState 上一个状态
	 */
	private void create(List<String> strs, List<Object> objs, SimpleIntArrayList states, SimpleIntArrayList checks,
SimpleObjectArrayList refObjs, int depth, int first, int last, int lastState)
	{
		if(first >= last)
			return;

		// base 的位置
		int base = 0;
		// 第一层进行特殊处理
		if(depth == 0)
		{
			base = lastState;	// 从上一个状态开始
			// 保证数组可以容纳本层结果
			int max = base + strs.get(last - 1).charAt(0) + 1;
			states.ensureCapacity(max);
			checks.ensureCapacity(max);
			refObjs.ensureCapacity(max);

			// 修改 checks 和 refObjs
			for(int i= first;i< last;i++)
			{
				String str = strs.get(i);
				char c = str.charAt(0);
				int len = str.length();
				int pos = base + c;		// 位置是 base + 字符
				checks.setQuick(pos, lastState);
				states.setQuick(pos, -1);	// 占位标记
				if(len == 1)	// 终态
				{
					refObjs.setQuick(pos, objs.get(i));
				}
			}
		}
		else
		{
			// 收集本层可能的字符
			List<Character> cs = new ArrayList<Character>();
			char lastChar = '\0';
			for(int i=first;i<last;i++)
			{
				char c = strs.get(i).charAt(depth);
				if(c != lastChar)
				{
					cs.add(c);
					lastChar = c;
				}
			}
			// 给这些字符安排一个位置
			if(cs.size() == 1)	// 对 1 特殊处理
			{
				char c = cs.get(0);
				if(m_scan < 1+ c)
					m_scan = 1 + c;
				int len = states.size();
				while(true)
				{
					if(m_scan >= len)
						break;
					else if(states.getQuick(m_scan) != 0)
						m_scan ++;
					else
						break;
				}
				base = m_scan - c;
			}
			else
			{
				double b = Math.log(states.size() + 1);
				b = b / (b + 1);
```

```
                        int begin = (int) (states.size() * b + 1); // 从 begin 开始寻找
                        int len = states.size();
                        for (int i = begin;; i++) // 因为 i 不断增加，所以，总能找到一个合适的位置
                        {
//                              scaned ++;

                                boolean isOk = true;
                                for (char c : cs)
                                {
                                        int pos = i + c;
                                        if (pos >= len) // 需要的位置已经超过 m_state 的大小了，说明已经满足要求了
                                        {
                                                break;
                                        }
                                        else if (states.getQuick(pos) != 0) // 位置已经被占用
                                        {
                                                isOk = false;
                                                break;
                                        }
                                }
                                if (isOk == false)
                                {
                                        continue;
                                }
                                else
                                {
                                        base = i;
                                        break;
                                }
                        }
                }
                // 保存对应的状态及参照对象
                int max = base + cs.get(cs.size() - 1) + 1;
                states.ensureCapacity(max);
                checks.ensureCapacity(max);
                refObjs.ensureCapacity(max);

                states.setQuick(lastState, base);
                for(char c:cs)
                {
                        int pos = base + c;
                        states.setQuick(pos, -1);
                        checks.setQuick(pos, lastState);
                }
                for(int i=first;i<last;i++)
                {
                        if(strs.get(i).length() == depth + 1)
                        {
                                refObjs.setQuick(base + strs.get(i).charAt(depth), objs.get(i));
                        }
                }
        }

        // 下一层的处理
        char lastChar = '\0';
        int f = -1;
        for(int i=first;i<last;i++)
        {
                String str = strs.get(i);
                if(str.length() > depth + 1)
                {
                        char c = str.charAt(depth);
                        if(lastChar != c)       // 如果当前字符不等于之前保存的字符
                        {
                                if(f >= 0)      // 如果之前还有没有处理的字符
                                        create(strs, objs, states, checks, refObjs, depth + 1, f, i, base + lastChar);
                                f = i;  // 保存本字符
                                lastChar = c;
                        }
                }
                else
                {
                        if(f >=0)       // 如果之前还有没有处理的字符
                                create(strs, objs, states, checks, refObjs, depth + 1, f, i, base + lastChar);
                        f = -1;
                        lastChar = '\0';
                }
        }
        if(f >=0)       // 处理最后的字符
                create(strs, objs, states, checks, refObjs, depth + 1, f, last, base + lastChar);
}
/**
 * 删除 key
 * <br>
 * 本处使用的算法不完整，仅仅删除了它最后一个字符对应的值，而没有删除整个值
 * <br>
 * 比如，"abc" 仅仅删除了 c，而"ab"还是占有位置。
```

```
     * <br>
     * 这样做的原因是为了加快删除速度。
     */
    public void delete(String key)
    {
        if(get(key) == null)
            return;

        int base = m_states.getQuick(1);
        int pos = 0;
        for(int i=0;i<key.length();i++)
        {
            char c = key.charAt(i);
            pos = base + c;
            base = m_states.getQuick(pos);
        }
        m_refObjs.setQuick(pos, null);
        if(m_states.getQuick(pos) <= 0)   // 如果是叶子状态就删除之
        {
            m_checks.setQuick(pos, 0);
            m_states.setQuick(pos, 0);
        }
    }
    /**
     * 设置 key 对应的对象
     */
    public void set(String key, T refObj)
    {
        if(get(key) == null)
            return;

        int base = m_states.getQuick(1);
        int pos = 0;
        for(int i=0;i<key.length();i++)
        {
            char c = key.charAt(i);
            pos = base + c;
            base = m_states.getQuick(pos);
        }
        m_refObjs.setQuick(pos, refObj);
    }
    /**
     * 得到所有关键词列表
     */
    public List<String> keys()
    {
        List<String> ret = new ArrayList<String>();
        for(int i=0;i<m_refObjs.size();i++)
        {
            if(m_refObjs.getQuick(i) != null)
            {
                StringBuilder str = new StringBuilder();
                int preState = m_checks.getQuick(i);
                int curState = i;
                while(preState != 0)
                {
                    int base = m_states.getQuick(preState);
                    str.append((char)(curState - base));
                    curState = preState;
                    preState = m_checks.getQuick(preState);
                }
                ret.add(str.reverse().toString());
            }
        }
        return ret;
    }
    /**
     * 得到值序列
     */
    public List<T> values()
    {
        List<T> ret = new ArrayList<T>();
        for(int i=0;i<m_refObjs.size();i++)
        {
            if(m_refObjs.getQuick(i) != null)
                ret.add((T)m_refObjs.getQuick(i));
        }
        return ret;
    }

    /**
     * 加入一个新的 key 到 Map 里面
     */
    public void put(String key, T refObj)
    {
        if(get(key) != null)
        {
```

```
        set(key,refObj);
        return;
}

//1, 找到最接近 key 的一个状态
int base = m_states.getQuick(1);
int pos = 0;
int lastPos = 1;
int depth = 0;
while(depth<key.length())
{
        char c = key.charAt(depth);
        pos = base + c;
        if(pos >= m_checks.size() || m_checks.getQuick(pos) != lastPos)
                break;
        base = m_states.getQuick(pos);
        lastPos = pos;
        depth ++;
}
//2, 把 key 融合到它里面去
if(depth >= key.length())    // 说明 key 这个字符串已经在状态中了，只是没有被 associate 上 object
{
        m_refObjs.setQuick(pos, refObj);
        return;
}
else
{
        if(base > 0)
        {
        char c = key.charAt(depth);
        if(base + c < m_checks.size())      // pos 还在空间内
        {
                if(m_states.getQuick(base + c) == 0)      // 位置未被占用
                {
                        // 使用这个 base 即可
                }
                else   // 位置被占用，需要搜寻新的位置
                {
                        // 1, 得到本层的所有值
                        List<Integer> vs = new ArrayList<Integer>();
                        int maxIndex = Math.min(0xffff, m_checks.size() - base);
                        for(int i=0;i<maxIndex;i++)
                        {
                                if(m_checks.getQuick(base + i) == lastPos)
                                {
                                        vs.add(i);
                                }
                        }
                        // 2, 搜寻新的位置
                        int newBase = 0;
                        double b = Math.log(m_states.size() + 1);
                        b = b / (b + 1);
                        int begin = (int) (m_states.size() * b + 1); //  从 begin 开始寻找
                        int len = m_states.size();
                        for (int i = begin;; i++) //  因为 i 不断增加，所以，总能找到一个合适的位置
                        {
                                boolean isOk = true;
                                for(Integer m: vs)
                                {
                                        int pos1 = i + m;
                                        if(pos1 >= len)
                                                break;
                                        else if(m_states.getQuick(pos1) != 0)
                                        {
                                                isOk = false;
                                                break;
                                        }
                                }
                                if(isOk)        // 新加入的字符串也必须满足
                                {
                                        int pos1 = i + c;
                                        if(pos1 < len && m_states.getQuick(pos1) != 0)
                                                isOk = false;
                                }
                                if (isOk == false)
                                {
                                        continue;
                                }
                                else
                                {
                                        newBase = i;
                                        break;
                                }
                        }
                        // 3, 刷新旧位置信息
                        int max = c + 1;
                        if(vs.size() > 0 && vs.get(vs.size() - 1) > max)
```

```
                        max = vs.get(vs.size( ) - 1);
                    max += newBase;
                    m_states.ensureCapacity(max);
                    m_checks.ensureCapacity(max);
                    m_refObjs.ensureCapacity(max);

                    m_states.setQuick(lastPos, newBase);
                    for(Integer m:vs)     // 把原来的数据都迁移过来
                    {
                        int oldPos = base + m;
                        int newPos = newBase + m;
                        int tBase = m_states.getQuick(base + m);
                        if(tBase > 0)
                        {
                            maxIndex = Math
                                    .min(0xffff, m_checks.size() - tBase);
                            for (int n = 0; n < maxIndex; n++)
                            {
                                int pos1 = tBase + n;
                                if (m_checks.getQuick(pos1) == oldPos)
                                {
                                    m_checks.setQuick(pos1, newPos);
                                }
                            }
                        }
                        // 拷贝旧数据
                        m_checks.setQuick(newPos, m_checks.getQuick(oldPos));
                        m_refObjs.setQuick(newPos, m_refObjs.getQuick(oldPos));
                        m_states.setQuick(newPos, m_states.getQuick(oldPos));
                        // 删除旧数据
                        m_states.setQuick(oldPos, 0);
                        m_checks.setQuick(oldPos, 0);
                        m_refObjs.setQuick(oldPos, null);
                    }
                    base = newBase;
                }
                pos = base + c;
                m_checks.setQuick(pos, lastPos);
                m_states.setQuick(pos, -1); //占有符号
            }
            else
            {
                int pos1 = base + c;
                m_states.ensureCapacity(pos1 + 1);
                m_checks.ensureCapacity(pos1 + 1);
                m_refObjs.ensureCapacity(pos1 + 1);

                // 加入新数据
                m_checks.setQuick(pos1, lastPos);
                m_states.setQuick(pos1, -1);
            }

            depth ++;
            lastPos = pos;
            }
            // 安排其余位置
            while(depth < key.length())
            {
                // 寻找一个位置
                char c1 = key.charAt(depth);
                m_scan = Math.max(m_scan, c1 + 1);
                while(true)
                {
                    if(m_scan < m_states.size())
                    {
                        if(m_states.getQuick(m_scan) == 0)       // 未使用
                            break;
                        else
                            m_scan ++;
                    }
                    else
                        break;
                }
                base = m_scan - c1;
                m_states.ensureCapacity(m_scan + 1);
                m_checks.ensureCapacity(m_scan + 1);
                m_refObjs.ensureCapacity(m_scan + 1);

                m_states.setQuick(lastPos, base);
                m_checks.setQuick(m_scan, lastPos);
                m_states.setQuick(m_scan, -1);    //占有符号
                depth ++;
                lastPos = m_scan;
            }
            m_refObjs.setQuick(lastPos, refObj);
        }
    }
```

```
/**
 * 清空数据
 */
public void clear()
{
        List<String> strs = new ArrayList<String>();
        List<T> objs = new ArrayList<T>();
        strs.add("goodzzp provider it!");
        objs.add(null);
        initial(strs, objs);
}
```

1.1.3 bit array(bitmap)

Bit 数组，里面的元素都是布尔值，而且，每个元素只占用 1 位。Bit 数组一般基于普通的 byte 或者 word 数组。

Bit 数组在实现上面，有大量的小技巧。这些笔者相信大家在很多 Bit 数组实现上都见过，此处不再赘述。如果在这个方面有什么疑问，可以参考：http://en.wikipedia.org/wiki/Bit_array。

Bit 数组也是一种重要的索引结构。它可以用来表示第 n 个文本是否包含了某个词语（数据）。有关它在索引上面的应用，在讲索引的章节会详细讨论。

1.1.4 Judy array

为什么要提及到 Judy array 这个超过 20000 行代码的咚咚？因为它在不降低速度的情况下，占用极小的额外空间！

Judy array 属于联合数组的范畴，看起来像一个 Trie 结构（或者说 256 叉树），即每个按照字节分层。它的主要目标是减少内存访问（官方说法是 avoid cache-line fill）。一般认为它有 3 个有点：

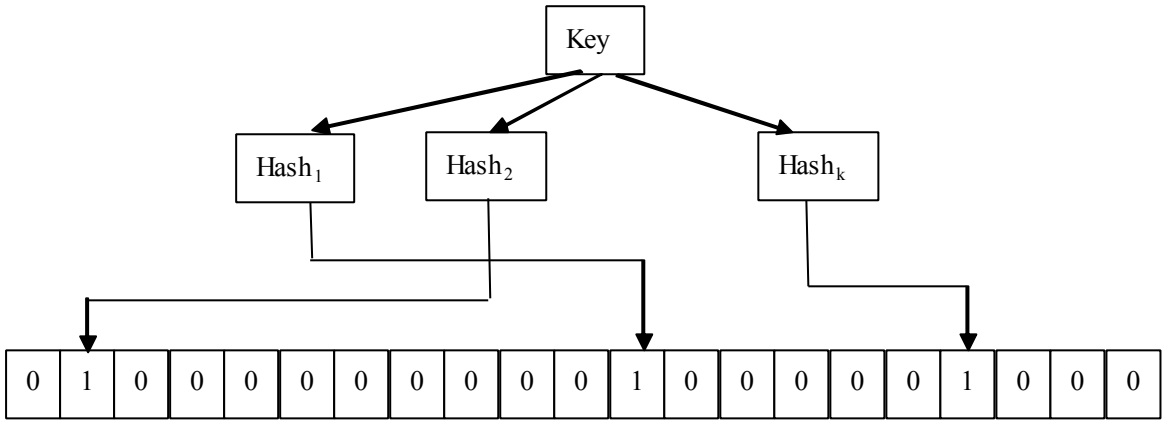
1) 空间占用小。这点是因为它是一个 Trie 结构，然后每层还利用了 Bitmap；
2) 速度快。Judy array 的主要贡献也就在此了，它减少了内存访问，自然就加快了计算速度。实际测试的结果表明，它在 add/update/delete 时的时间性能和 hashtable 相当（略慢）；
3) 比较抗算法复杂性攻击（Algorithms complex attacks）。算法复杂性攻击的一个典型例子就是经典快速排序（Intro QuickSort 的主要贡献就是解决了这个难题），如果你输入的是已经排序好了的数组，快排就变成噩梦了。同样，Hashtable 在输入的值的 hash 结果都一样的时候，也变得一无是处。

有论文对比过 Judy array 和 Hashtable 的性能（<key,vale>=<int,int>），发现前者一般只需占用 1.25 倍原始数据的空间，但 Hashtable 经常占到 3 倍-8 倍原始数据的空间。当然了，如果<key,value>对中的 value 占用空间很大，那么 Hashtable 和 Judy Array 占用的空间就差不多了。

1.1.5 Bloom filter

Bloom filter 是一个重要得没法说的结构。多年来，它都是大家研究的热点。Bloom filter 是一种高概率的 Set 结构，其主要特点是空间占用小。对于搜索引

擎来说，它在好几个地方都很有用处，比如 URL 去重、单词拼写检查等等。Bloom Filter 是一个概率 Set，只能以一定概率判断某个 Key 是否在 Set 里面出现。更准确的说，判断 Key 不在 Set 里面是精确的，判断 Key 在 Set 里面时不见得 Key 一定在 Set 里面，即所谓的 False Positive。下图说明了它的基本结构：



Bloom Filter 由若干个 Hash 函数和一个 bitmap 数组组成。其操作过程如下：

- 加入 Key：设置 bitmap 的第 $Hash_k(Key)$ 位为 1，这里 i=1, 2,… k；
- 查询 Key 是否存在：如果 bitmap 的第 $Hash_k(Key)$ 位均为 1，就认为 Key 在 Set 中存在。这里 i=1, 2,… k；
- 删除 Key：没有办法。

设 bitmap 有 m 位，而 Bloom Filter 集合有 n 个元素，那么，当 Bloom Filter 报 Key 存在而 Key 实际不存在的概率为：(1-(1-1/m)^(kn))^k，在 m 和 n 都很大的时候，这个结果可以为(1-e^(-kn/m))^k。同时，这个结果最小的时候，k 等于 m*ln2/n = 0.7*m/n，最小的值为(1/2^ln2)^(m/n) = 0.62^(m/n)。这里的 n 表示集合中元素的个数。这个式子表明，如果每个元素使用 10 位来表示的话（在 m 中，即 m = 10*n），bloom filter 出错的几率为 0.62^10 = 1%。

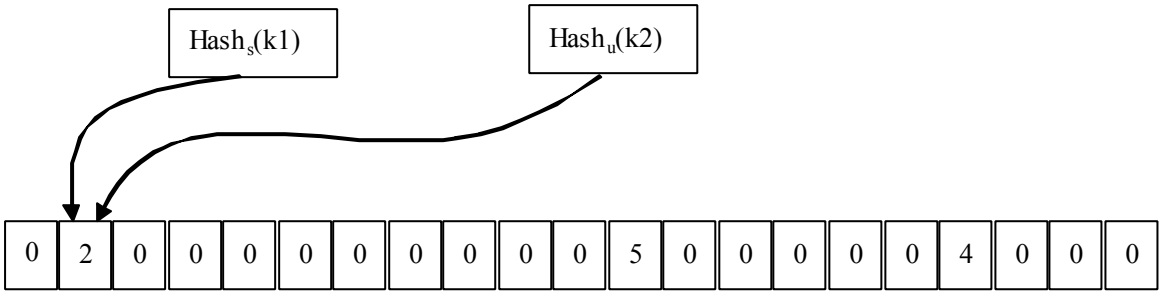
为了达到 p 的 false positive，每个元素需要 $1.44\log_2{}^{1/p}$ 位。

研究表明，当数据足够大时，如果用 kn 位表示 n 个数据的话，理论上，其 false positive 为：$P>=1/2^k-\varepsilon$，其中 ε 为常数。这就说明，理论上，要达到 p 的 false positive，每个元素只需要 $\log_2{}^{1/p}$ 位。可以看到，Bloom Filter 比理论值多用了 44%的空间。

显然，Bloom Filter 存在 2 个问题：1）空间利用率较低；2）需要使用很多独立分布的 Hash 函数；3）bloom filter 不支持删除操作。研究人员对此有很多改进，对几个主要的改进介绍如下：

- **增加对删除操作的支持（Counting Bloom filter）：**

如果数组的每个元素不是只有 1 位，而是有 t 位的话，这 t 位就可以用来保存有多少个 key 被映射到了这里。下图是一个例子：



也就是说，当key 被添加到 bloom filter 的时候，key 会被这 k 个 hash 映射到 k 个位置 pos1 到 posk，把这 k 个位置的值都加上 1。

在删除 key 的时候，key 会被这 k 个 hash 映射到 k 个位置 pos1 到 posk，把这 k 个位置的值都减 1。

研究证明，对于数组中的元素来说，它最后大于 2^4 的概率只有 6.78E-17。即每个元素只需要 4 位即可。

- **减少数组中为 1 的位（Choose Bloom Filter）：**

已经知道 Bloom filter 其实浪费了很多空间，在不减少它占用的空间的情况下，或许可以减少它 false positive 的概率。Choose Bloom Filter 试图减少数组中元素值为 1 的个数，它是这样计算 key 是否在 Set 中存在：计算 key 的 k 个 hash 值，设这 k 个 hash 值对应的 bitmap 的位中，为 1 的位数有 t 个，为 0 的位数为f 个（这里 k=t+f），当 t>f 的时候，就认为 key 在 Set 中存在，否则认为不存在。

如何决定需要设置为 0 的位（原来是 1），比较困难。

- **增加压缩比率（Compressed Bloom Filter）：**

Bloom Filter 常常需要保存在内存或者通过网络发送给客户端。但 Bloom Filter 在信息模型上是随机的，直接对它的压缩不够理想。Compressed Bloom Filter 的思想如下：

1）分配大一点的 bitmap；

2）按照基本 bloom filter 方式进行插入操作；

3）仔细选择，把部分为 0 的位设置为 1，增加压缩率。

假设原始 bitmap 为：

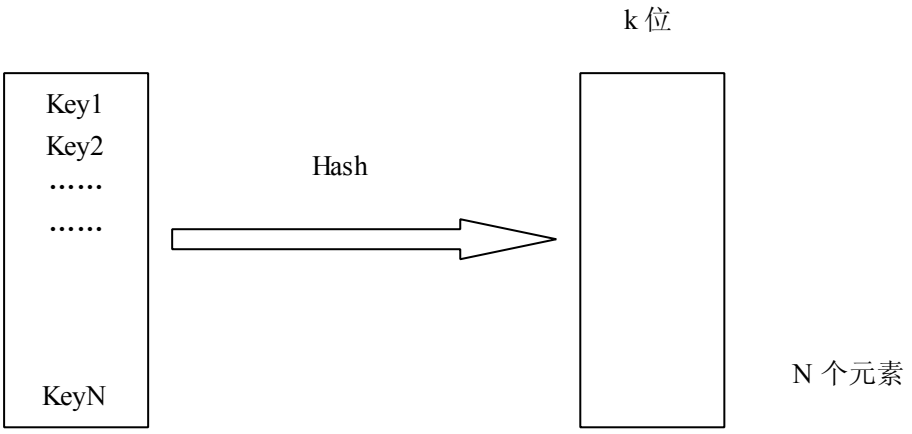
00000001000000100000000100000000000010000000

它可以被表示为：(7，6，8，11，7)，这里只保存连续的 0 的个数。现在有选择的把 0 改为 1，可以得到结果为（红色表示修改的位）：

00010001000100100010000100010001000100001000

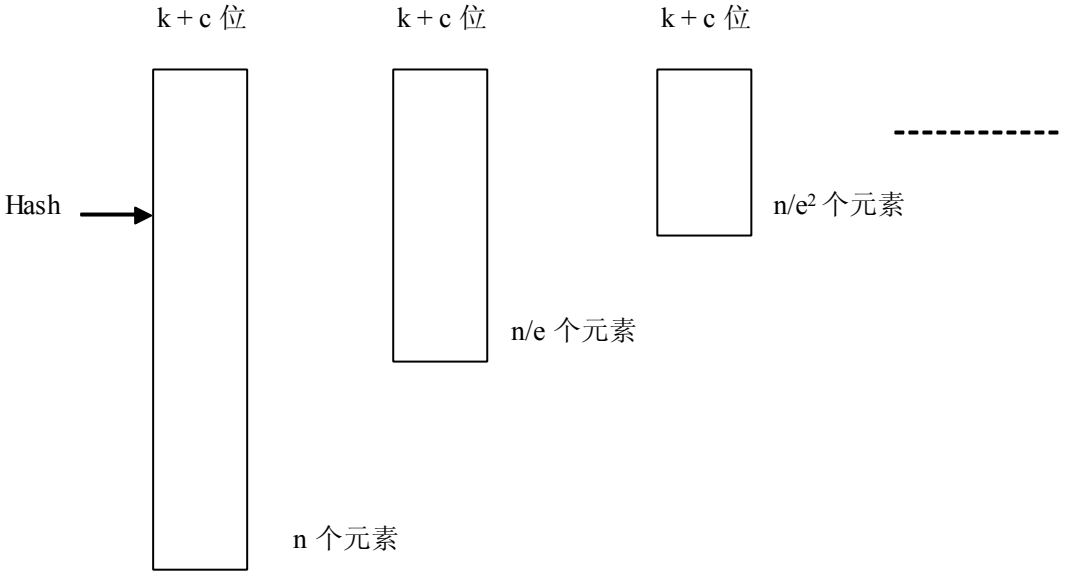
即为：（3，3，3，2，3，4，3，3，3，3，3），十分方便压缩。需要注意的是，因为增加了为 1 的位数，所以，Compressed Bloom Filter 比基本 Bloom Filter 更浪费空间。

Bloom Filter 的一种替代方案是直接使用 Hash。可以用完美 Hash（perfect hash）算法构建一个接近于理论值的概率 Set。其思想可以简单的表示为下图：



即通过完美 Hash，把输入的 N 个 Key，不冲突的映射到长度为 N 的数组 A，再把 Key 用 k 位表示即可。因为 k 位可以表示 2^k 种情况，故上述 Hash 模式可以达到的 false positive 为 $P >= 1/2^k$。

上面的模式可以扩展到普通 Hash 函数。比较有名的有 2 个，一个是用多数组解决 Hash 碰撞；一个就是 Cukoo Hash 了。



上图是基于多数组的 Hash Bloom Filter，其基本思想是，不碰撞的 Hash 放到第一个数组，碰撞的放到第二个数组，还有碰撞的继续放到下个数组。

Cukoo Hash 本来就是一个没有碰撞的 Hash，它不需要进行什么额外处理。唯一的问题在于如何在编程上更加节省空间。

1.2 压缩算法

压缩算法主要有 2 种：无损压缩和有损压缩。前者用来压缩文本和普通文件；后者一般用来压缩多媒体信息。因为搜索引擎很少用到有损压缩，我们将只谈论无损压缩。

1.2.1 无损压缩算法

压缩算法有很多种，大部分在我们平常已经接触过了，这里，笔者有选择的介绍一些，部分一笔带过。

- Run-length Encoding（RLE）：
  行程编码。行程编码的主要思想是把连续的重复字符用重复次数和重复字符表示。比如，“aaaaabb”用 RLE 可以表示为：5a 2b；
- Huffman Coding：
  霍夫曼编码是一种熵编码。在最小单位为 1 位的表示方法下，它是最优的。霍夫曼编码的实现可以参考笔者博客文章：Huffman 编码实现；
  - Adaptive Huffman Coding：
    自适应霍夫曼编码，是静态霍夫曼编码的变形。支持动态的添加数据；
  - T-Code：
    霍夫曼编码的一个变形；
- Arithmetic Coding：
  算术编码也是一种熵编码，它比霍夫曼编码更能接近熵的极限值，因为它可以用不到 1 位来表示一个数据。它是有专利限制的。算法编码的实现可以参考笔者博客文章：算术编码实例；
- Shannon-Fano Coding：
  Shannon-Fano 编码是很久前提出来的平衡二分法编码，效果比霍夫曼编码差。
- Range Encoding：
  范围编码的出现有点奇特，它是专门为了避免算术编码的专利而被设计出来的。在 32 位下，算术编码的范围为[0,2^{32}-1]，范围编码的范围为 [0，999999999]。
- LZ（Lempel-Ziv）系列：
  LZ 系列编码是基于字典的编码。它们的出现曾经让人大吃一惊。因为它们的特别重要性，这里对它们多介绍一点。
  LZ77 是最早提出来的 LZ 系列压缩算法。它使用一个滑动窗口来表示字典数据。LZ77 目前还使用在 defalte 算法、gzip 程序和 png 图片格式里面。LZ78 改进了字典，仅仅把不匹配的放入字典缓冲。LZW 算法继续改进了字典，它现在在 gif 图片和 compress 程序上都有应用。
  另外的几个常见的 LZ 系列算法包括：LZR（使用在 zip 格式上）、LZX（使用在 cab 格式上）。有 2 个重要的 LZ 系列算法值得特别关注，一个是 LZO 算法；一个是 LZMA 算法。澳大利亚人开发的 LZO 算法是已有的最快的压缩算法，其压缩速度接近于内存拷贝。LZMA 算法则是著名开源项目 7zip

的默认压缩算法，它的压缩效果一般都比商业软件 rar 格式好；

- Burrows-Wheeler transform(BWT)：
  BWT 编码是比较晚发现的算法（1994）。它通过重新排列字符串，让字符串变成容易压缩的序列。
- Prediction By Partial Matching(PPM)：
  部分匹配预测编码是目前为止发现的压缩文本最好的算法。
- 最近进展：
  最近出现了 3 种比较有名的新算法：WRT（Word Replacing Transform）、LZPM（LZ77 + PPM）和 LZC。有兴趣的读者可以自己去查资料。

1.2.2 整数编码

顾名思义，整数编码就是指编码对象是整数序列的压缩方法。其中，有一部分熵编码（Entropy Encoding）在整数编码上有大量应用，也放到这里进行介绍。

1.1.1.1熵编码

普通的无损压缩算法都是可以用来对整数序列进行压缩。人们常常使用算术编码或者 Huffman 编码来压缩整数序列。它们都是一种自适应的编码方式，即每个整数对应的编码根据整数序列而改变。还有一类熵编码，它们无论整数序列的分布如何，都按照固定的方式对整数进行编码，它们被叫做通用码（Universal code）。一般的说，通用码，只有在某种分布模型下，其编码结果才能达到熵最优。这种形式的编码有一个弱点：当整数序列和最优分布模型差异很大时，编码效果会很差。除了通用码类型的编码以外，还有一些其他基于熵编码的整数编码方式。

我们下面介绍几种常见的基于熵编码的整数编码方式：

- **Unary coding（一元编码）**
  一元编码使用 n-1 个 0 和一个 1 表示整数 n。它编码最优的时候，整数序列分布为：$P(n)=2^{-n}, n=1,2,3....$。这是因为 $H(p)=-\sum p(n)\log_2^{p(n)} \Leftrightarrow \sum p(n)\times n$。下面是几个编码实例：

n	Unary coding
1	1
2	0 1
3	00 1
4	000 1
5	0000 1
6	00000 1
7	000000 1
8	0000000 1
9	00000000 1

10	000000000 1

- **Elias gamma coding**

整数 n 被表示成 2 个部分，第一部分为$[\log_2^n]$个 0，第二部分为 n。其最优分布为笔者计算不出来（呜呜呜）。下面是几个编码实例：

n	Elias gamma coding
1	1
2	0 10
3	0 11
4	00 100
5	00 101
6	00 110
7	00 111
8	000 1000
9	000 1001
10	000 1011

- **Elias delta coding**

整数 n 被表示成 2 部分，第一部分是使用 gamma coding 表示的整数$1+[\log_2^n]$，第二部分是 n 的去掉最高位的 2 进制表示。下面是几个实际例子：

n	Elias delta coding
1	1
2	010 0
3	010 1
4	011 00
5	011 01
6	011 10
7	011 11
8	00100 000
9	00100 001
10	00100 010

- **Elias omega coding**

也叫 recursive Elias codes。它是循环的计算整数编码。给定整数 n，计算 Elias omega coding 的方式为：

1) 如果当前整数为 1，跳出；否则，写入整数的 2 进制表示到最前面；
2) 当前整数等于当前整数的位数减去 1；
3) 循环步骤 1 和 2；
4) 写一个 0 到末尾。

下面是几个例子：

n	Elias omega coding

1	0
2	10 0
3	11 0
4	10 100 0
5	10 101 0
6	10 110 0
7	10 111 0
8	11 1000 0
9	11 1001 0
10	11 1010 0

- **Exponential-Golomb coding**

整数 n 被编码为 $([\log_2^{(n+2^k)}]-k)$ 个 0 加上 $(n+2^k)$ 的二进制表示。这里，k 是自定义的变量，通常为 0。下面是几个例子：

n	k=0	k=1
0	1	10
1	0 10	11
2	0 11	0 100
3	00 100	0 101
4	00 101	0 110
5	00 110	0 111
6	00 111	00 1000
7	000 1000	00 1001
8	000 1001	00 1010
9	000 1010	00 1011
10	000 1011	00 1100

- **Fibonacci coding**

基于 Fibonacci（斐波那契）序列的编码。首先把整数 n 表示为 Fibonacci 序列值的和，即 $n=\sum d(k)F(k)$。其中，F(k)表示第 k 个 Fibonacci 数，d(k)为 0 或者 1，并且不存在连续的 d(k)和 d(k+1)都为 1。然后用 d(0)d(1)...d(m)1 表示 n，这里 d(m)必须为 1。

注意到 Fibonacci 序列为：{1,1,2,3,5,8,13,21,34,55,89,....}。

举例来说：

n	分解	Fibonacci coding
1	F(1)	1 1
2	F(2)	01 1
3	F(3)	001 1
4	F(1) + F(3)	101 1
5	F(4)	0001 1
6	F(1) + F(4)	1001 1

7	F(2) + F(4)	0101 1
8	F(5)	00001 1
9	F(1) + F(5)	10001 1
10	F(2) + F(5)	01001 1

- **Levenstein coding**

Levenstein 把整数 0 固定的编码为 0，对于其他数的编码计算如下：

1) C = 1;

2) 写入 n 的二进制表示，第一位不写入；

3) M = 第二步写入的二进制位数；

4) 如果 M 不为 0，就给 C 加上 1，让n=M，继续第二步；如果M 为 0，继续下一步；

5) 写入 C 个 1 和 1 个 0 到结果的最前面。

下面是几个例子：

n	Levenstein coding
0	0
1	10
2	110 0
3	110 1
4	1110 0 00
5	1110 0 01
6	1110 0 10
7	1110 0 11
8	1110 1 000
9	1110 1 001
10	1110 1 010

- **Golomb coding**

Golomb coding 是 1965 年提出来利用到 run-length 压缩里面的编码。它把整数 n 表示成 2 部分，第一部分是 n 除以 b 的商，第二部分是 n 除以 b 的余数。第一部分的表示是一元表示法(unary coding)，就是说，整数 m 被表示为 m 个 1 和一个 0。第二部分使用 truncated binary coding 表示法。下面是对 truncated binary coding 的简介：

truncated binary coding 的主要目的是减少位数的使用。对于 3 个数字来说，我们一般各需要 2 位来表示它们，即：

00：0；

01：1；

10：2；

11：unused；

truncated binary coding 主要利用没有使用的位来缩短其他位的长度。比如上面的可以变成：

0：0；

10：1；

11：2；

一种简单的计算 truncated binary coding 的方法为：

①现在要对 N 个数进行编码，编码的最大位数为 $r=[\log_2 N]+1$；r 位编码空闲的值有：$m = 2^r-N$ 个；

②对于第 i 个数（0<=i<=m），这样处理：a)分配第 2*i 个编码给它；b)去掉编码的最后一位。

③其他数字分派最后的未分配的编码。

如果想减少第一个数值的编码长度，可以通过增加前 m 个数其他数的编码长度来获得。比如，第一个数分配第一个编码减去后面$[\log_2(m+1)]$个数，后面的 N-1 个数分配余下的编码。

举例来说，当 b 为 3 的时候，0 表示为：00；1 表示为：010；2 表示为：011。这里，余数 b 的选择十分关键，有人建议使用 0.69 * average(A)来表示数组 A。另外 Rice coding 是 golomb coding 的变形，它要求 b 为 2 的幂次。

Golomb coding 的最佳分布是几何分布 $P(n) = p^n(1-p)$。其中当 $b = \log_p^{0.5}$ 时，Golomb 编码达到最好效果。下面是几个例子：

n	p=0.5，b=1	p=0.7，b=2	p=0.794，b=3	p=0.84，b=4	p=0.95，b=14
0	0	0 0	0 0	0 00	0 000
1	10	0 1	0 10	0 01	0 001
2	110	10 0	0 11	0 10	0 0100
3	1110	10 1	10 0	0 11	0 0101
4	11110	110 0	10 10	10 00	0 0110
5	111110	110 1	10 11	10 01	0 0111
6	1111110	1110 0	110 0	10 10	0 1000
7	11111110	1110 1	110 10	10 11	0 1001
8	111111110	11110 0	110 11	110 00	0 1010
9	1111111110	11110 1	1110 0	110 01	0 1011
10	11111111110	111110 0	1110 10	110 10	0 1100

1.1.1.2可变字节编码（variable byte code）

可变字节编码是一种应用十分普遍的压缩整数的编码。它使用每个字节的最

高位表示标记，余下的 7 位表示数据。多个连续字节表示一个完整的数据。

举例来说，9 ＝ $(1001)_2$，所以，9 表示为：$(10001001)_2$。这里，最高位为 1 表示最后一个字节。而 135 = (1 10000111)$_2$，所以，135 表示为：$(00000001)_2$ $(10000111)_2$。其编码的方式就是把输入分解为 7 位 7 位的，每个 7 位用一个字节表示。

这种编码的主要特点是一个整数仍然被编码为多个字节，十分方便处理。以下是几个例子：

n	variable byte code
0	10000000
1	10000001
2	10000010
3	10000011
127	11111111
128	00000001 10000000
129	00000001 10000001

显然，对于 4 字节的整数来讲，可变字节编码最多能够压缩到 1/4 的大小。

1.1.1.3二元插入编码（binary interpolative coding）

二元插入编码是专门用来编码有序整数序列的。假设我们的输入是一个单调递增整数序列，对于序列中的任意 2 个值 $n_i, n_j (i < j)$ 来说，当知道 n_i, n_j 后，我们就知道了 $n_k (i < k < j)$ 的取值范围。范围越小意味着用来表示值的位数越小。具体来说，这个算法这么操作：

1) 假设我们有如下 7 个值需要编码：{3，8，9，11，12，13，17}；而且我们知道这些值的范围在[1－20]。
2) 如果我们知道了这些偶数位的数：{8，11，13}，那么第一个数的范围为 1－7，需要 3 位既可编码；第三个数范围为 10－11，需要 1 位即可编码；第五个数字范围为 12，需要 0 位编码；第 6 个数字范围为 14－19，需要 3 位编码。
3) 递归的来说，如果我们知道了序列{8，11，13}的偶数位数字 11 的话，我们就能按照上面的方式对奇数位数据编码。
4) 递归进行到最后只会留下一个数字。这里是 11，这个数字的范围在 1－19，所以，它需要 5 位即可编码。

二元插入编码的具体实现，参考笔者的博客文章：二元插入编码的实现。