目录

C++ primer 补充内容 1

第 2 章 变量和基本类型

1. 常量表达式是指值不会改变并且编译过程就能得到计算结果的表达式。
2. C++11 允许将变量声明为 constexpr 类型以便编译器来验证变量是否是一个常量表达式。

```
Constexpr int mf=20;                    // mf 是常量表达式
constexpr int limit=mf+1;               // mf+1 是常量表达式
constexpr int sz=size();        // 只能当 size 是一个 constexpr 函数时，才是一条正确的声明语句
```

   不能使用普通函数作为 constexpr 变量的初始值。

3. 在 constexpr 声明中如果定义了一个指针，限定符 constexpr 仅对指针有效，与指针所指的对象无关。

```
Const int *p=nullptr;                   // p 是一个指向整形常量的指针
constexpr int *q=nullptr;               // q 是一个指向整数的常量指针
```

第 3 章 字符串、向量和数组

1. 位于头文件的代码一般来说不应该使用 using 声明。因为头文件的内容会拷贝到所有引用它的文件中去，如果头文件里有某个 using 声明，那么每个使用了该头文件的文件就都会有这个声明。
2. string 对象的初始化方式：

- string s1

  空串。

- String s2(s1)

  s2 是 s1 的副本。

- String s2=s1

  等价于 s2(s1)。

- String s3("value")

  s3 是字面值“value”的副本。

- String s3 = "value"

  等价于 s3("value")，s3 是字面值“value”的副本。

- String s4(n,'c')

  把 s4 初始化为由连续 n 个字符 c 组成的串。

3. string 的操作：

- os<<s

将 s 写到输出流 os 当中，返回 os。

- is>>s

  从 is 中读取字符串赋给 s，字符串以空白分隔，返回 is。

- getline(is,s)

  从 is 中读取一行赋给 s，返回 is。

- s.empty()

  判断 s 是否为空。

- s.size()

  返回 s 中字符的个数。

- s[n]

  返回 s 中第 n 个字符的引用，位置 n 从 0 计起。

- s1+s2

  返回 s1 和 s2 连接后的结果。

4. 标准库允许把字符字面值和字符串字面值转换成 string 对象，所以在需要 string 对象的地方可以使用字面值来替代：

```
String s1="hello",s2="world";
string s3=s1+","+s2+'\n';
```

必须确保每个加法运算符的两侧至少有一个是 string：

```
String = s1+",";                    // 正确
string = "hello"+",";               // 错误，两个对象都不是 string
```

5. C++11 提供了范围 for 语句。这种语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，语法形式是：

```
for(declaration:expression)
    statement
```

expression 表示的必须是一个序列，比如用花括号括起来的初始值列表，数组或 vector、string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员。这个跟 python 里面的 for 作用是一样的，例如：

```
String str("some string");
for(auto c:str)
    cout<<c<<endl;
```

6. 初始化 vector 对象的方法：

- vector<T> v1

  v1 是一个空 vector，它潜在的元素是 T 类型，执行默认初始化。

- vector<T> v2(v1)

  v2 中包含 v1 所有元素的副本。

- vector<T> v2 = v1

  等价于 v2(v1)，v2 中包含 v1 所有元素的副本。
- vector<T> v3(n,val)

  v3 包含了 n 个重复的元素，每个元素的值都是 val。
- vector<T> v4(n)

  v4 包含了 n 个重复地执行了值初始化的对象，值初始化为 0。
- vector<T> v5{a,b,c...}

  v5 包含了初始值个数的元素，每个元素被赋予相应的初始值。
- vector<T> v5={a,b,c...}

  等价于 vector<T> v5{a,b,c...}。

7. vector 支持的操作：

- v.empty()

  判断 v 是否为空。
- v.size()

  返回 v 中元素的个数。
- v.push_back(t)

  向 v 的尾端添加一个值得为 t 的元素。
- v[n]

  返回 v 中第 n 个位置上元素的引用。
- v1={a,b,c...}

  用列表中元素的拷贝替换 v1 中的元素。

8. 不能使用下标添加元素：

```
for(decltype(ivec.size() ix=0;ix!=10;++ix))
    ivec[ix]=ix;                    // 不能这样使用下标添加元素，应该使用 push_back()
```

9. 所有的标准库容器都可以使用迭代器。这些类型都拥有名为 begin 和 end 的，其中 begin 成员负责返回指向第一个元素（或第一个字符）的迭代器。end 成员则负责返回指向容器“尾元素的下一位置”的迭代器：

```
Auto b=v.begin(),e=v.end();
```

10. 标准窗口迭代器的操作：

- *iter

  返回迭代器 iter 所指元素的引用。
- iter->mem

  解引用 iter 并获取该元素名为 mem 的成员，等价于(*iter).mem

- ++iter

  令 iter 指示容器中的下一个元素。

- --iter

  令 iter 指示容器中的上一个元素。

- iter1==iter2

  判断两个迭代器是否相等。

11. begin 和 end 返回的具体类型由对象是否是常量决定，如果对象是常量，begin 和 end 返回 const_iterator；如果对象不是常量，返回 iterator；

```
vector<int> v;
const vector<int> cv;
auto it1=v.begin();                    // it1 的类型是 vector<int>::iterator
auto it2=cv.begin();                   // it2 的类型是 vector<int>::const_iterator
```

12. vector 和 string 迭代器支持的运算：

- iter+n

迭代器加上一个整数值仍得一个迭代器，迭代器指示的新位置与原来相比向前移动了若干元素。

- iter-n

  迭代器指示的位置与原来相比向后移动了 n 个元素。

- Iter1 – iter2

  两个迭代器相减的结果是它们之间的距离。

13. C++11 新标准引入了 begin 和 end 两个新函数，两个函数与容器中的两个同名函数功能类似，返回相应的迭代器：

```
Int ia[]={0,1,2,3,4,5,6};
int *beg=begin(ia);
int *last=end(ia);
```

14. C 风格字符串的函数：

- strlen(p)

  返回 p 的长度，空字符不计算在内。

- strcmp(p1,p2)

  如果 p1==p2，则返回 0；如果 p1>p2，返回一个正值；如果 p1<p2，返回一个负值。

- strcat(p1,p2)

  将 p2 附加到 p1 之后，返回 p1。

- strcpy(p1,p2)

  将 p2 拷贝给 p1，返回 p1。

15. 多维数组的初始化：

```
Int ia[3][4] = {
    {0,1,2,3},
    {4,5,6,7},
    {8,9,10,11},
};
```

内层嵌套的花括号不是必需的，也可以使用下面这条语句定义，效果是一样的：

```
Int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

16. 如果仅想初始化第一行的第一个元素，通过如下语句即可：

```
Int ia[3][4]={{0},{4},{8}};
```

其他未列出的元素执行默认初始化。

17. 范围 for 可以使用到数组和多维数组中：

```
size_t cnt=0;
for(auto &row:ia)                    // 对于外层数组的每一个元素
    for(auto &col:row){              // 对于内层数组的每一个元素
        col=cnt;
        ++cnt;
}
```

注意，由于需要改变 ia 的值，所以要把 row 和 col 声明为引用，否则程序无法通过编译。

18. 多维数组名字也会自动将其转换成指向首元素的指针：

```
for(auto p=ia;p!=ia+3;++p){
    for(auto q = *p;q!*p+4;++q)
        cout << *q<<' ';
    cout<<endl;
}
```

第 **5** 章 语句

1. 在 case 子句中定义变量时，如果该变量在另一个 case 中使用，则不能直接进行初始化：

```
 int i=0;
  switch (i) {
  case 1:
        int m=0;                    // 出错，不能直接在这里初始化
        int m;                      // 正确
        m=0;                       // 语句正确，但编译器应该会忽略这个定义，因为输出并不是 0
        cout << "this is cout 1"<<endl;
        break;
```

```
  default:
        cout << m<<endl;
        break;
  }
```

第 6 章 函数

1. 函数的形参可以为空，为了与 C 语言兼容，可以使用关键字 void 表示没有形参：

```
Void v2(void){}
```

2. 为了编写能处理不同数量实参的函数，C++11 提供了两种方法：

- 如果所有实参类型都相同，可以传递一个名为 initializer_list 的标准库类型。
- 使用可变参数模板。

3. initializer_list 是一种标准库类型，用于表示某种特定类型的值的数组。initializer_list 提供的操作：

- initializer_list<T> lst;

  默认初始化；T 是元素的空列表。

- initializer_list<T> lst{a,b,c...};

  lst 的元素数量和初始值一样多；lst 的元素是对应初始值的副本；列表中的元素是 const。

- lst2(lst)

  lst2=lst

  拷贝或赋值一个 initializer_list 对象，但不会拷贝列表中的元素。拷贝后，原始列表和副本共享元素。

- Lst.size()

  列表中的元素数量。

- Lst.begin()

  返回指向 lst 中首元素的指针。

- lst.end()

  返回指向 lst 中尾元素下一位置的指针。

4. initializer_list 也是一种模板类型。

```
initializer_list<string> ls;
```

和 vector 不同的是，initializer_list 对象中的元素永远是常量值，无法修改。

5. 如果想向 initializer_list 形参传递一个值的序列，则必须把序列放在一对花括号内：

```
Void error_msg(initializer_list<string> il)
{
```

```
    // 函数体
}
if(expected!=actual)
    error_msg({"functionX",expected,actual});
else
    error_msg({"functionX","okay"});
```

6. 数组不能被拷贝，所以函数不能返回数组，但可以返回数组的指针或引用。
7. constexpr 函数是指能用于常量表达式的函数。定义 constexpr 函数的方法与其他函数类似，不过 constexpr 函数的返回类型及所有形参的类型都得是字面值类型，而且函数体中必须有且只有一条 return 语句：

```
Constexpr int new_sz(){return 42;}
constexpr int foo=new_sz()
```

8. 为了在编译过程中随时展开，constexpr 函数被隐式地指定为内联函数。

第 7 章 类

1. 构造函数的名字和类名相同，但构造函数没有返回值。

```
Class Sales_data{
    Sales_data()=default;
    sales_data(const std::string &s):bookNo(s){}
}
```

- 在 C++新标准中，如果我们需要默认的行为，那么可以通过在参数列表后面写上 =default 来要求编译器生成构造函数。其中，=default 既可以和声明一起出现在类的内部，也可以作为定义出现在类的外部。
- bookNo(s)称为构造函数初始值列表，负责为新创建的对象的一个或几个数据成员赋初值。

2. 在类外定义成员函数时首先写的是返回值类型：

```
Class Screen{
public:
    int add(int x,int y);
}

int Screen::get(int x,int y) const
{
    // 函数的定义
}
```

3. 令成员函数作为友元时，必须明确指出该成员函数属于哪个类：

```
Class Screen{
    friend void Window_mgr::clear(ScreenIndex);
```

```
};
```

4. 在类中，如果成员使用了外层作用域中的某个名字，而该名字代表一种类型，则类不能在之后重新定义该名字：

```
Typedef double money;
class account{
public:
    money balance(){return bal;}          // 使用外层的 money
private:
    typedef double money;
    money bal;                            // 错误，不能重新定义 money
}
```

5. C++11 扩展了构造函数，使用用户可以定义委托构造函数。所谓的委托构造函数就是使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把自己的一些职责委托给其他构造函数。

```
Class sales_data{
    public:
        sales_data(std::string s,unsigned cnt,double price):
                bookNo(s),units_sold(cnt),revenue(con*price){}
        // 下面这两个构造函数将工作都委托给最上面那个构造函数
        sales_data():sales_data("",0,0){}
        sales_data(std::string s):sales_data(s,0,0){}
}
```

6. 聚合类具有特殊的初始化语法形式，用户可以直接访问其成员。聚合类条件：

- 所有成员都是 public。
- 没有定义构造函数。
- 没有类内初始值。
- 没有基类，也没有 virtual 函数。

```
Struct Data{
    int ival;
    string s;
};
```

我们可以提供一个花括号的成员初始值列表，用它来初始化聚合类的数据成员：

```
Data vall = {0,"Anna"};
```

初始值的顺序必须与声明顺序一致。

7. 尽管构造函数不能是 const 的，但是字面值常量类的构造函数可以是 constexpr。事实上，一个字面常量类必须至少提供一个 constexpr 构造函数。

第 8 章 IO 库

1. cin 是 istream 类型对象，而 cout 是 ostream 类型对象。
2. IO 库类型和头文件：

头文件	类型
iostream	istream，wistream，从流读取数据
	ostream，wostream，向流写入数据
	iostream，wiostream，读写流
fstream	ifstream，wifstream，从文件读取数据
	ofstream，wofstream，向文件写入数据
	fstream，wfstream，读写文件
stringstream	istringstream，wistringstream，从 string 读取数据
	ostringstream，wostringstream，向 string 写入数据
	stringstream，wstringstream，读写 string

3. IO 对象不能拷贝或赋值。
4. IO 库条件状态：

- strm::badbit

  strm::badbit 用来指出流已崩溃。
- Strm::failbit

  strm::failbit 用来指出一个 IO 操作失败了。
- Strm::eofbit

  strm::eofbit 用来指出流到达了文件结束。
- Strm::goodbit

  strm::goodbit 用来指出流未处于错误状态。此值保证为零。
- s.eof()

  若流的 eofbit 置位，则返回 true。
- s.fail()

  若流 s 的 failbit 或 badbit 置位，则返回 true。
- s.bad()

  若流 s 的 badbit 置位，则返回 true。
- s.good()

  若流 s 处于有效状态，则返回 true。
- s.clear()

  若流 s 中所有条件状态位复位，将流的状态设置为有效。返回 void。
- s.clear(flags)

根据给定的 flags 标志位，将流 s 中对应条件状态位复位。

- s.setstate(flags)

  根据给定的 flags 标志位，将流 s 中对应条件状态位置位。

- s.rdstate()

  返回流 s 的当前条件状态。

5. fstream 特有的操作

- fstream fstrm;

  创建一个未绑定的文件流。

- Fstream fstrm(s);

  创建一个 fstream，并打开名为 s 的文件。

- Fstream fstrm(s,mode);

  按指定的 mode 模式创建一个 fstream。

- Fstrm.open(s);

  打开名为 s 的文件，并将文件与 fstrm 绑定。

- Fstrm.close();

  关闭 fstrm 绑定的文件。返回 void。

- Fstrm.is_open();

  返回一个 bool 值，指出与 fstrm 关联的文件是否成功打开且尚未关闭。

6. 每个流都有一个关联的文件模式：

- in：以读方式打开。
- out：以写方式打开。以该模式打开文件会丢弃已有数据。
- app：每次写操作前均定位到文件末尾。
- ate：打开文件后立即定义到文件末尾。
- trunc：截断文件。
- binary：以二进制方式进行 IO。

第 9 章 顺序容器

1. 顺序容器类型：

- vector：可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢。
- deque：双端队列。支持快速随机访问。在头尾位置插入删除速度很快。
- list：双向链表。只支持双向顺序访问。
- forward_list：单向链表。只支持单向顺序访问。在链表任何位置插入删除都很快。

- array：固定大小数组。支持快速随机访问。不能添加或删除元素。
- string：与 vector 相似的容器，但专门用于保存字符。随机访问快。在尾部插入删除速度很快。

2. array 是固定大小数组，不支持所有能改变大小的相关操作。
3. 假设 c 是一种容器，容器相关操作：

- c.size();

  c 中元素的数目（forward_list 不支持）。
- c.max_size();

  c 可保存的最大元素数目。
- c.empty();

  若 c 中存储了元素，返回 false，否则返回 true。
- c.swap(d);

  交换 c 和 d 的元素。
- swap(c,d);

  交换 c 和 d 的元素。
- c.insert(args);

  将 args 中的元素插入 c。不适用于 array。
- c.emplace(inits);

  使用 inits 构造 c 中的一个元素。不适用于 array。
- c.erase(args);

  删除 args 指定的元素。不适用于 array。
- c.clear();

  清空 c。不适用于 array。

4. 顺序容器都支持 begin 和 end 等迭代器函数。
5. 向顺序容器添加元素：

- c.push_back(t);

  c.emplace_back(args);

  将 t 或由 args 创建的元素添加到 c 的尾部。
- c.push_front(t);

  c.emplace_front(args);

  将 t 或由 args 创建的元素添加到 c 的头部。
- c.insert(p,t);

  c.emplace(p,args);

  将 t 或由 args 创建的元素插入迭代器 p 指向的元素之前。返回指向新添加的元素的迭

代器。

- c.insert(p,n,t);

  在迭代器 p 指向的元素之前插入 n 个值为 t 的元素。

- c.insert(p,b,e);

  将迭代器 b 和 e 指定的范围内的元素插入到迭代器 p 指向的元素之前。

- c.insert(p,il);

  il 是一个花括号包围的元素值列表。将这些值插入到迭代器 p 指向的元素之前。

➢ 注意：array 不支持所有插入操作；forward_list 有自己的专有版本的 insert 和 emplace，而且不支持 push_back 和 emplace_back；vector 和 string 不支持 push_front 和 emplace_front。

6. 在顺序容器中访问元素：

- c.back();

  返回 c 中尾元素的引用。不能用于 forward_list。

- c.front();

  返回 c 中首元素的引用。

- c[n];

  返回 c 中下标为 n 的元素的引用。只适用于 string、vector、deque 和 array。

- c.at(n);

  返回下标为 n 的元素的引用。只适用于 string、vector、deque 和 array。

7. 顺序容器的删除操作：

- c.pop_back();

  删除 c 中尾元素。forward_list 不支持该操作。

- c.pop_front();

  删除 c 中首元素。vector 和 string 不支持该操作。

- c.erase(p);

  删除迭代器 p 所指定的元素，返回一个指向被删元素之后元素的迭代器，若 p 指向尾元素，则返回尾后迭代器。

- c.erase(b,e);

  删除迭代器 b 和 e 所指定范围内的元素。

- c.clear();

  清空 c 中的所有元素。

8. 顺序窗口大小操作：

- c.resize(n);

  将 c 的大小调整为 n。

- c.resize(n,t);

  将 c 的大小调整为 n，任何新加的元素都初始化为 t。

9. 向容器中添加元素和从容器中删除元素可能会使容器元素的指针、引用或迭代器失效。也就是说，当改变容器元素大小时，相关的指针等已经变化，从而导致指针、迭代器失效。
10. 构造 string 的其他方法：

- string(cp,n);

  创建一个 string 对象，使用 cp 指向的数组中的前 n 个字符来初始化。

- string(s2,pos2);

  创建一个 string 对象，使用 s2 从下标 pos2 开始的字符来初始化。

- string(s2,pos2.len2);

  创建一个 string 对象，使用 s2 从下标 pos2 开始的 len2 个字符来初始化。

11. string 类还定义了 append()和 replace()两个函数，这两个函数可以改变 string 的内容。
12. 容器适配器事实上也是一种容器，它使用基本容器来实现，它改变接口而不改变底层实现方式。
13. 除了顺序容器，标准库还提供了三种顺序容器适配器：queue（先进先出的队列）、priority_queue（优先队列） 和 stack。 事实上，这个适配器应该跟 deque 一样是容器类。

- A a;

  创建一个名为 a 的空适配器。

- A a(c);

  创建一个名为 a 的适配器，带有容器 c 的一个拷贝。

- a.empty()

  若 a 包含任何元素，则返回 false。

- a.size();

  返回 a 中元素的数目。

- Swap(a,b);

  a.swap(b);

  交换 a 和 b 的内容。

```
Stack<int> stk(deq);                    // 从 deq 拷贝元素到 stk
```

14. 栈的操作：

- s.pop();

  删除栈顶元素，但不返回该元素。

- s.push(item);

s.emplace(args);

将 item 或 args 中的元素压入栈顶。

- s.top();

返回栈顶元素，但不将元素弹出栈。

15. queue 和 priority_queue 的操作：

- q.pop();

返回 queue 的首元素或 priority_queue 的最高优先级的元素，但不删除此元素。

- q.front();

返回首元素或尾元素，但不删除此元素。

- q.top();

返回最高优先级元素，但不删除元素。只适用于 priority_queue。

- q.push(item);

q.emplace(args);

将 item 或由 args 构造的值插入 queue 末尾或 priority_queue 中恰当的位置。

第 10 章 泛型算法

1. Iterator 操作相关函数：

(1) begin()：返回指向第一个元素的 iterator 指针。

(2) end：用法与 begin 同，通常是一起使用来遍历容器中的元素。返回指向最后一个元素再后面一个的 iterator 指针。

(3) rbegin：返回指向最后一个元素的 reverse_iterator 指针。

(4) rend：返回指向第一个元素前面一个元素的 reverse_iterator 指针，通常与 rbegin 一起使用来反向遍历容器。

(5) cbegin、cend：返回值类型为 const_iterator，功能同 begin、end。

(6) crbegin、crend：返回值类型为 const_reverse_iterator，功能同 crbegin、crend。

2. 一般情况下，泛型算法并不直接操作容器，而是遍历由两个迭代器指定的一个元素范围来进行操作。

3. find 算法用于查找给定的值：

```
int val = 42;
auto result = find(vec.cbegin(),vec.cend(),val);
cout << "The value " << val<<endl;
```

前两个参数是表示元素范围的迭代器，第三个是要查找的值。

4. accumulate 算法返回累加的结果：

```
Int sum = accumulate(vec.cbegin,vec.cend,0);
```

第三个参数是和的初值。

5. equal 算法用于确定两个序列是否相同：

```
equal(roster1.cbegin(),roster1.cend(),roster2.cbegin());
```

6. 使用写容器元素算法时必须确保序列有足够的空间。

7. 算法 fill 接受用于向指定容器写入指定数据：

```
fill(vec.begin(),vec.end(),0);
```

将每个元素重置为 0。与 fill()不同，fill_n()可以指定重置元素的个数：

```
fill_n(vec.begin(),vec.size(),0);
```

将 vec 元素指定位置指定个数的值重置为 0。

8. 插入迭代器是向容器中添加元素的迭代器。插入迭代器用法如下：

```
vector<int> vec;
auto it = back_inserter(vec);
*it=42;
```

9. 可以使用 back_inserter 创建一个迭代器，来提供给泛型算法使用：

```
vector<int> vec;
fill_n(back_inserter(vec),10,0);
```

添加 10 个元素到 vec，每个元素赋值为 0。

10. 拷贝算法接受三个迭代器，前两个表示一个输入范围，第三个表示目的序列的起始位置，此类算法将输入范围中的元素拷贝到目的序列中：

(1) copy 算法

```
Int a1[]={0,1,2,3,4,5,6};
int a2[sizeof(a1)/sizeof(*a1)];                    // 两者的大小一样
auto ret = copy(begin(a1),end(a1),a2);
```

把 a1 的内容拷贝给 a2。

(2) replace 算法接受 4 个参数，前两个是迭代器，表示输入序列，第三个是要搜索的值，第四个是新值：

```
replace(ilst.begin(),ilst.end(),0,42);
```

将指定位置的 0 替换为 42。

(3) replace_copy 算法和 replace 算法不同之处在于，replace_copy 算法原来的序列保持不变，将新替换的序列保存到另一个地址：

```
replace(ilst.begin(),ilst.end(),back_inserter(ivec),0,42);
```

调用后，ilst 并未改变，ivec 包含 ilst 的一份副本，不过原来在 ilst 中值为 0 的元素在 ivec 中都变是 42。

11. sort 算法用于对输入序列中的元素进行排序，它是利用元素< 运算符来实现的。

12. unique 算法用于消除重复出现的元素。

13. 谓词是一个可调用的表达式，其返回结果是一个能用作条件的值。标准库算法所使用的谓词分为两类：一元谓词（只接受单一参数）和二元谓词（有两个参数）。接受谓词参数的算法对输入序列中的元素调用谓词。接受一个二元谓词参数的 sort 版本用这

个谓词代替<来比较元素。

```
Bool isShorter(const string &s1,const string &s2)
{
      return s1.size() < s2.size();
}
sort(words.begin(),sords.end(),isShorter);
```

此调用会将 words 重排，使得所有长度为 3 的单词排序长度为 4 的单词之前，然后是长度为 5 的单词，依此类推。

14. stable_sort 算法和 sort 算法类似，只是该算法会在按长度大小重排的基础上，相同长度的元素按字典排序。

15. lambda 表达式可以理解为匿名函数。

```
[ capture-list ] ( params list )  ->  return type {function body }
```

其中 capture-list：捕获的局部变量列表，通常为空。

```
[](const string &a,const string &b){return a.size() < b.size();}
```

参数列表和返回类型可以忽略：

```
Auto f = []{return 42;};
```

16. 调用 lambda 的调用普通函数一样。

17. 可以把 lambda 当作谓词来使用：

```
stable_sort(words.begin(),sords.end(),[](const string &a,const string &b){return a.size() <
b.size();})
```

18. 引入 lambda 表达式的前导符是一对方括号，称为 lambda 引入符。这个引入符的作用就是表明，其后的 lambda 表达式以何种方式使用这些变量。

- []：不捕获任何外部变量
- [=]：以值的形式捕获所有外部变量
- [x]：x 以值的形式捕获
- [&]：以引用形式捕获所有外部变量
- [x, &y]： x 以传值形式捕获，y 以引用形式捕获
- [=, &z]：z 以引用形式捕获，其余变量以传值形式捕获
- [&, x]：x 以值的形式捕获，其余变量以引用形式捕获

```
float f0 = 1.0f;
float f1 = 10.0f;
std::cout << [=, &f0](float a) { return f0 += f1 + std::abs(a); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

输出是 14.5 和 14.5。在这个例子中，f0 通过引用被捕获，而其它变量，比如 f1 则是通过值被捕获。

19. for_each 算法接受一个可调用对象，并对输入序列中每个元素调用此对象：

```
for_each(wc,words.end(),[](const string &s){cout<<s<<" ";});
cout<<endl;
```

打印长度大于等于给定单词的词，每个单词后面接一个空格

20. 默认情况下，对于一个被拷贝的变量，lambda 不会改变其值。如果希望能改变一个被捕获的变量，就必须在参数列表首加上 mutable：

```
Auto f = [v1] () mutable{rerurn ++v1;};
```

21. 使用标准库的 bind()将普通函数转化为 lambda 可使用的可调用对象，bind()的调用一般形式为：

```
Auto newCallable = bind(callable,arg_list);
```

newCallable 是 lambda 可使用的可调用对象，callable 是一个函数，arg_list 是需要传递给 callable 的参数。当我们使用 newCallable 时，newCallable 会调用 callable 函数，并传入 arg_list 参数。

```
Bool check_size(const string &s,string::size_type sz)
{
    return s.size() >= sz;
}

auto check6 = bind(check_size,_1,6)
```

22. 默认情况下，bind 的那些参数被拷贝到 bind 返回的可调用对象中。但是对于那些以引用方式传递，或者无法拷贝的类型，必须使用标准库 ref 函数：

```
for_each(word.begin(),words.end(),bind(print,ref(os),_1,' '));
```

函数 ref 返回一个对象，包含给定的引用，此对象是可以拷贝的。而 cref 函数和 ref 函数类型，它生成一个保存 const 引用的类。这两个函数在头文件 functional 中。

23. C++ 语言还提供了另外三种迭代器：

- 插入迭代器：这类迭代器与容器绑定在一起，实现在容器中插入元素的功能。
- iostream 迭代器：这类迭代器可与输入或输出流绑定在一起，用于迭代遍历所关联的 IO 流。
- 反向迭代器：这类迭代器实现向后遍历，而不是向前遍历。所有容器类型都定义了自己的 reverse_iterator 类型， 由 rbegin 和 rend 成员函数返回。
- 移动迭代器：这些迭代器用于移动其中的元素。

24. 插入迭代器接受一个容器，生成一个迭代器，能实现向给定容器添加元素。插入迭代器操作：

-

```
it=t
```

在 it 指定的当前位置插入值 t。对于普通的插入器，it= value 等效于如下操作：

```
pos = container.insert(pos,value);
++pos;
```

即在指定位置插入元素，同时更新下次插入位置。

•

```
*it,++it,it++
```

这个操作不对 it 做任何可情。每个操作都返回 it。

25. C++ 语言提供了三种插入器，其差别在于插入元素的位置不同：

- back_inserter，在尾部插入元素，创建使用 push_back 实现插入的迭代器。
- front_inserter，在首部插入元素，使用 push_front 实现插入。
- inserter，使用 insert 实现插入操作。除了所关联的容器外，

26. 流迭代器 istream_iterator 用于读取输入流，而 ostream_iterator 则用于写输出流。

27. iostream 迭代器的构造函数

- istream_iterator<T> in(strm);

  创建从输入流 strm 中读取 T 类型对象的 istream_iterator 对象。

- istream_iterator<T> in;

  istream_iterator 对象的超出末端迭代器。

- ostream_iterator<T> in(strm);

  创建将 T 类型的对象写到输出流 strm 的 ostream_iterator 对象

- ostream_iterator<T> in(strm, delim);

创建将 T 类型的对象写到输出流 strm 的 ostream_iterator 对象，在写入过程中使用 delim 作为元素的分隔符。delim 是以空字符结束的字符数组（C 风格字符串）。

```
Ifstream in("afile");
istream_iterator<int> int_it(in);
```

从文件 afile 获取 int 型值。

28. istream_iterator 的操作：

(1) it1 == it2;

it1 != it2;

比较两上 istream_iterator 对象是否相等（不等）。迭代器读取的必须是相同的类型。如果两个迭代器都是 end 值，则它们相等。对于两个都不指向流结束位置的迭代器，如果它们使用同一个输入流构造，则它们也相等。

(2) *it;

返回从流中读取的值。

(3) it->mem;

是 (*it).mem 的同义词。返回从流中读取的对象的 mem 成员。

(4) ++it;

it++;

通过使用元素类型提供的 >> 操作从输入流中读取下一个元素值，使迭代器向前移动。

通常，前缀版本使用迭代器在流中向前移动，并返回对加 1 后的迭代器的引用。而后缀版本使迭代器在流中向前移动后，返回原值。

29. ostream_iterator 的操作：

(1) out=val;

用<<运算符将 val 写入到 out 所绑定的 ostream 中。

(2) *out,++out,out++

这些运算符不对 out 做任何事。

30. 反向迭代器是一种反向遍历容器的迭代器。也就是，从最后一个元素到第一个元素遍历容器。反向迭代器将自增（和自减）的含义反过来了：对于反向迭代器，++ 运算将访问前一个元素，而 － 运算则访问下一个元素。

```
vector<int> vec={0,1,2,3,4,5,6,7,8,9};
for(auto r_iter=vec.crbegin(),r_iter!=vec.crend();++r_iter)
      cout<<*riter<<endl;
```

31. 在任何其他算法分类之上，还有一组参数规范。大多数算法具有如下 4 种形式之一：

```
alg(beg,end,other args);
alg(beg,end,dest,other args);
alg(beg,end,beg2,other args);
alg(beg,end,beg2,end2,other args);
```

- alg：算法名字。
- beg 和 end：算法俄输入范围。
- dest、beg2 和 end2：都是迭代器参数，分别指定目的位置和第二范围。

32. 接受谓词参数的算法都附加有_if 前缀，如 find_if()。

33. 排序算法中，将排序结果复制到其他空间的算法在名字后面附加一个_copy，如 reverse_copy()，该算法按逆序排序并复制到参数 dest 指定的空间中。

34. 一些算法同时提供_copy 和_if，这些版本接受一个目的位置迭代器和一个谓词参数：

```
remove_copy_if(v1.begin(),v1.end(),back_inserter(v2),[](int i){rerutn i%2;});
```

35. 通用的 sort 要求随机访问迭代器，所以不适合链表类型容器 list 和 forward_list。它们定义了独有的 sort、merge、remove、reverse 和 unique：

- list.merge(list2)：将来自 list2 的元素合并到 list。list 和 list2 都必须是有序的。
- List.remove(val)：删除与给定值相等或令一元谓词为真的每个元素。
- List.remove_if(pred)： 删除令一元谓词为真的每个元素。
- List.reverse()：逆序排序。
- List.sort()：排序。
- List.unique()：删除相同的值。

第 12 章 关联容器

1. 关联容器中的元素是按关键字来保存和访问的，而顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。
2. 关联容器类型：

- map：关联数组；保存键-值对。
- set：只保存键值对中的键的容器。
- multimap：关键字可重复出现的 map。
- multiset：关键字可重复出现的 set。
- unordered_map：用哈希函数组织的 map。
- unordered_set：用哈希函数组织的 set。
- unordered_multimap：哈希组织的 map；关键字可重复出现
- unordered_multiset：哈希组织的 set；关键字可重复出现

```
map<string,size_t> word_count;
string word;
while(cin>>word)
      ++word_count[word];
for(const auto &w:word_count)
      // 打印结果
      cout << w.first<<"occurs"<<w.second<<((w.second>1)?"times":"time")<<endl;
```

使用关联数组的单词计数程序。

3. pair 上的操作：

- pair<T1,T2> p;

  创建一个 pair。
- pair<T1,T2> p(v1,v2);

  创建一个 pair，first 和 second 成员分别用 v1 和 v2 进行初始化。
- pair<T1,T2> p={v1,v2};

  等价于 p(v1,v2)。
- make_pair(v1,v2);

  返回一个由 v1 和 v2 初始化的 pair。pair 的类型从 v1 和 v2 的类型推断出来。
- p.first;

  返回 p 的名为 first 的数据成员。
- p.second;

  返回 p 的名为 second 的数据成员。

4. 通常不应该对关联容器使用泛型搜索算法。
5. 关联容器也支持迭代器。

```
Auto map_it = word_count.cbegin();
```

6. 关联容器的 insert()向容器中添加一个元素或一个元素范围：

```
Vector<int> ivec={2,4,6,8};
set<int> set2;
set2.insert(ivec.cbegin(),ivec.cend());
set2.insert({1,3,5,7});
```

上述就是 insert 的两个版本，分别打接受一对迭代器，或是一个初始化器列表。

7. 对于允许重复的关键字，接受单个元素的 insert 操作返回一个指向新元素的迭代器。
8. 使用 erase()来从关联容器中删除指定元素。该函数接受一个迭代器或一个迭代器对或一个元素范围为参数。
9. 在一个关联容器中查找元素：

- c.find(k);

  查找指定的元素，返回指向第一个关键字的迭代器。

- c.count(k);

  计算指定元素的数量。

- c.lower_bound(k);

  返回一个迭代器，指向第一个关键字不小于 k 的元素。

- c.upper_bound(k);

  返回一个迭代器，指向第一个关键字大于 k 的元素。

- c.equal_range(k);

  返回一个迭代器 pair，表示关键字等于 k 的元素的范围。若 k 不存在，pair 的两个成员等于 c.end()。

10. 新标准定义了 4 个无序关联容器，这些容器不使用<来组织元素，而是使用一个哈希函数和关键字类型的==运算符。通常在关键字类型的元素没有明显的序关系的情况下，无序容器是非常有用的。
11. 无序容器在存储上组织为一组桶，每个桶保存零个或多个元素。无序容器使用一个哈希函数将元素映射到桶。为了访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶。
12. 无序容器管理操作：

- c.bucket_count();

  正在使用的桶的数目。

- c.max_bucket_count();

  容器能容纳的最多的桶的数量。

- c.bucket_size(n);

  第 n 个桶中有多少个元素。

- c.bucket(k);

  关键字为 k 的元素在哪个桶。

- c.load_factor();

  每个桶的平均元素数量，返回 float 值。

- c.max_load_factor();

  c 试图维护的平均桶的大小，返回 float 值。c 会在需要时添加新的桶，以使用 load_factor<=max_load_factor。

- c.rehash(n);

  重组存储，使得 bucket_count>=n 且 bucket_count>size/max_load_factor。

- c.reserve(n);

  重组存储，使得 c 可以保存 n 个元素且不必 rehash。

13. 无序容器迭代器：

- local_iterator：可以用来访问桶中元素的迭代器类型。
- const_local_iterator：桶迭代器的 const 版本。
- c.begin(n),c.end(n)：桶 n 的首元素迭代器和尾后迭代器。
- c.cbegin(n),c.cend(n)：和前两个函数类似，但返回 const_local_iterator。

第 12 章 动态内存

1. 用 new 来分配 const 对象是合法的：

```
Const int *pci=new const int(1024);
```

2. 当内存耗尽时，new 表达式会失败，系统抛出一个类型为 bad_alloc 的异常。使用 nothrow 可以阻止抛出异常：

```
Int *p2=new (throw) int;
```

3. 使用类似下列的语句释放动态数组：

```
Delete [] pa;
```

4. allocator 类用于将分配内存，但和 new 表达式不同的是，它并不构造对象。也就是说，对于没有默认构造函数的类，不能使用 new 来分配内存，但可以使用 allocator 来分配内存。使用 allocator 类可以实现定义的内存分配器。

```
allocator<string> alloc;                    // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);           // 分配 n 个未初始化的 string
```

5. allocator 类及其算法：

- allocator<T> a;

  定义了一个名为 a 的 allocate 对象，它可以为类型为 T 的对象分配内存。

- a.allocate(n);

  分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象。

- a.deallocate(p,n);

  释放从 T*指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。

- a.construct(p,args);

  将 args 传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象。

- a.destroy(p);

  p 是个 T*类型的指针，对 p 指向的对象执行析构函数。

6. allocator 算法用于在未初始化的内存中创建对象：

- uninitialized_copy(b,e,b2);

  从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 中的元素必须足够大，能容纳输入序列中元素的拷贝。

- uninitialized_copy_n(b,n,b2);

  从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中。

- uninitialized_fil(b,e,t);

  在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝。

- uninitialized_fill_n(b,n,t);

  从迭代器 b 指向的内存地址开始创建 n 个对象。

第 13 章 拷贝控制

1. 要新标准下，我们可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数来阻止拷贝。删除的函数虽然声明了，但不能以任何方式使用它们。在函数的参数列表后面加上=delete 来指出我们希望将它定义为删除的：

```
Struct NoCopy{
    NoCopy()=default;              // 使用合成的默认构造函数
    NoCopy(const NoCopy&)=delete;                  // 阻止拷贝
    NoCopy &operator=(const NoCopy&)=delete;       // 阻止赋值
}
```

2. 与=default 不同，=delete 必须出现在函数第一次声明的时候。

3. 对于某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的，则类的合成析构函数被定义为删除的。

- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。

也就是说，如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数将被定义为删除的。

4. 如果具有引用成员或无法默认构造的 const 成员的类，编译器不会为其合成默认构造函数。
5. 如果一个类有 const 成员，则它不能使用合成的拷贝赋值运算符。
6. 标准库定义了一个名为 swap 的函数，这个函数用于交换两个元素的值。如果类定义了自己的 swap 来交换两个对象的值，则调用时会使用自定义的 swap 而不是标准库的 swap。
7. 为了支持移动操作，新标准引入了一种新的引用类型——右值引用。所谓右值引用就是必须绑定到右值的引用。通过&&而不是&来获得右值引用。事件上，右值引用也是某个对象的别名。
8. 右值引用绑定到一个将要销毁的对象，因此，可以自由地将一个右值引用的资源“移动”到另一个对象中。
9. 常规引用不能将引用绑定到表达式、字面常量或返回右值的表达式。右值引用完全相反，可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上。

```
Int i=42;
int &r=i;                       // 正确，r 引用 i
int &&rr=i;                     // 错误：不能将一个右值引用绑定到一个左值上
int &r2=i*42;                   // 错误：i*42 是一个右值
const int &r3=i*42;                 // 正确：可以将一个 const 的引用绑定到一个右值上
int &&rr2=i*42;                      // 正确：将 rr2 绑定到乘法结果上
```

左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

10. 标准库提供了一个名为 move 的标准库函数来获得绑定到左值的右值引用。

```
Int &&rr3=std::move(rr1);
```

11. 移动构造函数是移动资源的函数。移动构造函数的第一个参数是该类类型的右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。移动构造函数还必须确保销毁原来的对象：

```
Strvec::strvec(strvec &&s) noexcept:elements(s.elements),first_free(s.first_free),cap(s.cap)
{
    s.elements = s.first_free=s.cap=nullptr;
}
```

上面的移动构造函数在构造函数初始化列表中获取元素的值并保存，然后在函数体中将原来的对象销毁。

12. 移动构造函数通常要声明为 noexcept，即不抛出异常。
13. 移动赋值运算符执行与析构函数和移动构造函数相同的工作。

```
Strvec &strvec::operator=(strvec &&rhs) noexcept
{
    if(this!=&rhs){
        free();                         // 释放已有元素
        elements=rhs.elements;          // 从 rhs 接管资源
        first_free=rhs.first_free;
        cap=rhs.cap;
        // 将 rhs 置于可析构状态
        rhs.elements=rhs.first_free=rhs.cap=nullptr;
    }
    return *this;
}
```

14. 与处理拷贝构造函数一样，编译器也会合成移动构造函数和移动赋值运算符。

第 15 章 面向对象程序设计

1. C++新标准提供了一种防止继承的方法，即在类名后跟一个关键字 final，用来禁止继承自该类。

```
Class NoDerived final;
class Base{/*   */};
class Last final:base{/*    */};
```

2. 静态类型：对象在声明时采用的类型，在编译期即已确定；

   动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的；

   静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；

   动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；

```
class B
{
    /* 函数体  */
}
class C : public B
{
    /* 函数体  */
}
class D : public B
{
    /* 函数体  */
}

D* pD = new D();                    // pD 的静态类型是它声明的类型 D*，动态类型也是 D*
```

```
B* pB = pD;                    // pB 的静态类型是它声明的类型 B*，动态类型是 pB 所指向的对象
                               // pD 的类型 D*
C* pC = new C();
pB = pC;                       //pB 的动态类型是可以更改的，现在它的动态类型是 C*
```

3. 静态绑定和动态绑定的区别：

- 静态绑定发生在编译期，动态绑定发生在运行期；
- 对象的动态类型可以更改，但是静态类型无法更改；
- 要想实现动态，必须使用动态绑定；
- 在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定；

4. 定义虚函数是为了允许用基类的引用或指针来调用子类的这个函数。
5. 引用或指针的静态与动态类型不同是 C++支持多态性的根本所在。
6. 当我们使用基类的引用或指针调用基类中定义的一个虚函数时，直到运行时才决定到底执行哪个版本，判断依据是引用或指针所绑定的对象的真实类型。
7. 一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。
8. 派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，编译器将认为新定义的函数与基类中原有的函数是相互独立的。
9. 有时候，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数某个特定版本。使用作用域运算符实现这一目的：

```
Double undiscounted = baseP->Quote::net_price(42);
```

强制使用 Quote 的 net_price 函数，而不管 baseP 实际指向的对象类型到底是什么。

10. 名字查找先与类型检查，也就是说，即使函数形参不相同，在派生类中定义的函数也会覆盖基类的成员。正因为如此，虚函数和派生类函数的形参必须完全一样。
11. 如果派生类希望所有重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或一个也不覆盖。
12. 继承关系对基类拷贝控制最直接的影响就是基类通常应该定义一个虚析构函数，以确定执行正确的析构函数版本。

```
Class Quote{
public:
    virtual ~Quote()=default;
}
```

只要基类的析构函数是虚函数，就能确保当我们 delete 基类指针时将运行正确的析构版本：

```
Quote *itemP = new Quote;          // 静态类型与动态类型一致。
delete itemP;                      // 调用 Quote 的析构函数
itemP = new Bulk_quote;            // 静态类型与动态类型不一致
delete itemP;                      // 调用 bulk_quote 的析构函数
```

注意：上面的例子 Bulk_quote 是 Quote 的子类。如果基类的析构函数不是虚函数，则 delete itemP 一句将产生未定义的行为。

第 16 章 模板与泛型编程

1. 在代码中包含函数模板本身不会生成函数定义，它只是一个用于生成函数定义的方案。编译器使用模板为特定类型生成函数定义时，得到的是模板实例。
2. 除了定义类型参数，还可以在模板中定义非类型参数。一个非类型参数表示一个值而非一个类型。

```
template<unsigned N,unsigned M>
int compare(const char (&p1)[N],const char (&p2)(M))
{
    return strcmp(p1,p2);
}
```

第一个模板参数 N 表示第一个数组的长度，第二个参数表示第二个数组的长度。调用 compare()：

```
compare("hi","mom");
```

调用后，编译器会使用字面常量的大小来代替 N 和 M，从而实例化模板。

3. 在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用非类型参数，例如，指定数组大小。
4. 函数模板可以推断模板类型，但类模板不能推断参数类型。

```
cout<<compare(1,0)<<endl;
```

函数会推断出模板类型为 int。

5. 当一个类模板调用另一个模板时，通常不将实际类型的名字用作其模板实参。相反，通常将模板自己的参数当作被使用的实参：

```
Std::shared_ptr<std::vectir<T> > data;
```

6. 当我们使用一个类模板类型时必须提供模板实参，但有一个例如：在类模板自己的作用域中，我们可以直接使用模板名而不提供实参：

```
Template <typename T> class BlobPtr{
public:
    BlobPtr& operator++();
    BlobPtr& operator--();
}
```

这里使用的是 BlobPtr&而不是 BlobPtr<T>&。

7. 在类个使用类模板名：

```
Template <typename T> class BlobPtr{
public:
    BlobPtr& operator++();
    BlobPtr& operator--();
}
```

```
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int){
    /* 函数体  */
}
```

8. 在模板内不能重用模板参数名：

```
Typedef double A;
template <typename A,typename B> void f(A a, B b)
{
    A temp=a;                    // 模板参数 A 可以覆盖外部变量 A，正确。
    double B;                    // 错误，不允许重用模板参数 B
}
```

9. 允许提供默认模板实参：

```
Template <typename T,typename F=less<T> >
int compare(const T &v1,const T &v2, F f=f())
{
    /* 函数体  */
}
```

10. 当两个或多个独立编译的源文件使用了相同的模板，并提供相同的参数时，每个文件中就都会有该模板的一个实例。在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。

    可以通过 extern 来避免这种开销。

```
extern template class Blob<string>;                    // 声明
```

11. 函数模板可以被另一个模板或普通非模板函数重载。
12. 一个可变参数模板就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为参数包。存在两种参数包：

- 模板参数包：表示零个或多个模板参数。
- 函数参数包：表示零个或多个函数参数。

13. 用一个省略号来指出一个模板参数或函数参数表示一个包。

```
Template <typename T,typename ...  Args>
void foo(const T &t,const Args&... rest);
```

14. 使用 sizeof...()来获取参数包中的元素数量：

```
template<typename ... Args> void g(Args ... args){
    cout<<sizeof...(Args)<<endl;          // 类型参数的数目
    cout<<sizeof...(args)<<endl;          // 函数参数的数目
}
```

第 17 章 标准库特殊设施

1. tuple 是类似 pair 的模板。tuple 元组定义了一个有固定数目元素的容器，其中的每个元素类型都可以不相同，这与其他容器有着本质的区别。

2. tuple 支持的操作：

(1) tuple<T_1,T_2,...,T_n> t;

定义一个 tuple，成员数为 n，第 i 个成员的类型是 T_i。

(2) tuple<T_1,T_2,...T_n> t(v_1,v_2,...v_n);

定义一个 tuple，每个成员用对应的初始值 v_i 来进行初始化。此构造函数是 explicit 的。

(3) make_tuple(v_1,v_2,...v_n)

返回一个用给定初始值初始化的 tuple。tuple 的类型从初始值的类型推断。

(4) t_1==t_2

t_1 !=t_2

当两个 tuple 具有相同数量的成员且成员对应相等时，两个 tuple 相等。

(5) t_1 relop t_2

tupler 的关系运算使用字典序。两个 tuple 必须具有相同数据的成员。

(6) get<i>(t)

返回 t 的第 i 个数据成员的引用；如果 t 是一个左值，结果是一个左值引用，否则，结果是一个右值引用。

(7) tuple_size<tupleType>::value

一个类模板，可以通过该模板 value 成员获取 tuple Type 类型对象的元素数量。

(8) tuple_element<i,tupleType>::type

一个类模板，type 是该模板成员，返回 tupleType 第 i 个元素的数据类型。

3. 当我们定义一个 tuple 时，需要指出每个成员的类型：

```
tuple<size_t,size_t,size_t> threeD;
tuple<string,vector<double>,int,list<int>> someVal("constants",{3.14,2.178},42,{0,1,2,3})
```

4. 使用一个名为 get 的标准库模板访问 tuple 的成员：

```
Auto book=get<0>(item);
auto cnt=get<1>(item);
```

尖括号中的值必须是一个整形常量表达式，不能是变量。

5. decltype 与 auto 关键字一样，用于进行编译时类型推导，decltype 的作用是选择并返回操作数的数据类型。和 auto 不同的是，decltype 不实际计算表达式的值：

```
int i = 4;
```

```
decltype(i) a; //推导结果为 int。a 的类型为 int。
```

decltype 用法：

- 推导出表达式类型。如上面的例子所述。
- 与 using/typedef 合用，用于定义类型。

```
using size_t = decltype(sizeof(0));//sizeof(a)的返回值为 size_t 类型
using ptrdiff_t = decltype((int*)0 - (int*)0);
using nullptr_t = decltype(nullptr);
```

- 重用匿名类型：

```
struct
{
    int d ;
    doubel b;
}anon_s;

decltype(anon_s) as ;//定义了一个上面匿名的结构体
```

- 泛型编程中结合 auto，用于追踪函数的返回值类型：

```
template <typename _Tx, typename _Ty>
auto multiply(_Tx x, _Ty y)->decltype(_Tx*_Ty)
{
    return x*y;
}
```

这也是 decltype 最大的用途了。

6. 如果不知道 tuple 的准确类型信息，可以使用两个辅助类模板来查询 tuple 成员的数量和类型：

```
Typedf decltype(item) trans;

/* 返回 trans 类型对象成员的数量 */
size_t sz=tuple_size<trans>::value;

/* cnt 类型与 item 中第二个成员相同 */
tuple_element<1,trans>::type cnt=get<1>(item)
```

7. tuple 的常见用途是返回多个不同类型的值。
8. bitset 类提供了一种抽象方法来操作位的集合。初始化 bitset 的方法：

- bitset<n> b;

  b 有 n 位，每一位均为 0。此构造函数是一个 constexpr。

- Bitset<n> b(u);

b 是 unsigned long long 数值 u 的低 n 位的拷贝。如果 n 大于 unsigned long long 的大小，则 b 中超出的高位被置为 0。此构造函数是一个 constexpr。

```
Bitset<32> bitvec(1U);              //1U 是一个数值，表示 unsigned long long 型的数值 1
```

32 位，低位为 1，其他位为 0。

- bitset<n> b(s,pos,m,zero,one);

b 是 string s 从位置 pos 开始 m 个字符的拷贝。s 只能包含字符 zero 或 one；如果 s 包含任何其他字符，构造函数会抛出 invalid_argument 异常。

- bitset<n> b(cp,pos,m,zero,one);

与上一个函数相同，从 cp 指向的字符数据中拷贝字符。如果未提供 m，则 cp 必须指向一个 C 风格的字符串。如果提供 m，则从 cp 开始必须至少有 m 个 zero 或 one 字符。

```
Bitset<13> bitvec1(oxbeef);
```

二进制序列为 1111011101111。

9. 我们可以从一个 string 或一个字符数组来初始化 bitset。

```
Bitset<32> bitvec("1100");
```

2、3 位为 1，其余为 0。

10. bitset 操作定义了多种检测或设置一个或多个二进制位的方法：

- b.any()：b 中是否存在置位的二进制位。
- b.all()：b 中所有位都置位了吗？
- b.count()：b 中置为 1 的二进制位的个数。
- b.size()：b 中二进制位的个数。
- b[pos]：访问 b 中在 pos 处的二进制位。
- b.test(pos)：b 中在 pos 处的二进制位是否为 1？
- b.set()：把 b 中所有二进制位都置为 1
- b.set(pos)：把 b 中在 pos 处的二进制位置为 1
- b.reset()：把 b 中所有二进制位都置为 0
- b.reset(pos)：把 b 中在 pos 处的二进制位置为 0
- b.flip()：把 b 中所有二进制位逐位取反
- b.flip(pos)：把 b 中在 pos 处的二进制位取反
- b.to_ulong()：用 b 中同样的二进制位返回一个 unsigned long 值
- os << b：把 b 中的位集输出到 os 流

```
bool is_set = bitvec.any();          // false，所有位都是 0
bool is_not_set = bitvec.none();     // true，所有位都是 0
```

11. regex 类表示一个正则表达式。相关的组件：

- basic_regex：正则表达式对象，是一个通用的模板，有 typedef basic_regex<char> regex 和 typedef basic_regex<char_t>wregex。
- regex_match：将一个字符序列和正则表达式匹配。

- regex_search：寻找字符序列中的子串中与正则表达式匹配的结果，在找到第一个匹配的结果后就会停止查找。
- regex_replace ：使用格式化的替换文本，替换正则表达式匹配到字符序列的地方。
- regex_iterator ：迭代器，用来匹配所有的子串。
- match_results：容器类，保存正则表达式匹配的结果。
- sub_match：容器类，保存子正则表达式匹配的字符序列.

12. C++ regex 函数有 3 个：regex_match、 regex_search 、regex_replace。

(1) regex_match()：将一个字符序列和正则表达式匹配。

```
#include <iostream>
#include <string>
#include <regex>

int main ()
{

  if (std::regex_match ("subject", std::regex("(sub)(.*)") ))
    std::cout << "string literal matched\n";

  std::string s ("subject");
  std::regex e ("(sub)(.*)");
  if (std::regex_match (s,e))
    std::cout << "string object matched\n";

  if ( std::regex_match ( s.begin(), s.end(), e ) )
    std::cout << "range matched\n";

  std::cmatch cm;    // same as std::match_results<const char*> cm;
  std::regex_match ("subject",cm,e);
  std::cout << "string literal with " << cm.size() << " matches\n";

  std::smatch sm;    // same as std::match_results<string::const_iterator> sm;
  std::regex_match (s,sm,e);
  std::cout << "string object with " << sm.size() << " matches\n";

  std::regex_match ( s.cbegin(), s.cend(), sm, e);
  std::cout << "range with " << sm.size() << " matches\n";

  // using explicit flags:
  std::regex_match ( "subject", cm, e, std::regex_constants::match_default );

  std::cout << "the matches were: ";
  for (unsigned i=0; i<sm.size(); ++i) {
```

```
    std::cout << "[" << sm[i] << "] ";
  }

  std::cout << std::endl;

  return 0;
}
```

(2) regex_search()：寻找字符序列中的子串中与正则表达式匹配的结果，在找到第一个匹配的结果后就会停止查找。

```
#include <iostream>
#include <regex>
#include <string>

int main(){
  std::string s ("this subject has a submarine as a subsequence");
  std::smatch m;
  std::regex e ("\\b(sub)([^ ]*)");   // matches words beginning by "sub"

  std::cout << "Target sequence: " << s << std::endl;
  std::cout << "Regular expression: /\\b(sub)([^ ]*)/" << std::endl;
  std::cout << "The following matches and submatches were found:" << std::endl;

  while (std::regex_search (s,m,e)) {
    for (auto x=m.begin();x!=m.end();x++)
      std::cout << x->str() << " ";
    std::cout << "--> ([^ ]*) match " << m.format("$2") <<std::endl;
    s = m.suffix().str();
  }
}
```

(3) regex_replace()：使用格式化的替换文本，替换正则表达式匹配到字符序列的地方。

```
#include <regex>
#include <iostream>

int main() {
    char buf[20];
    const char *first = "axayaz";
    const char *last = first + strlen(first);
    std::regex rx("a");
    std::string fmt("A");
    std::regex_constants::match_flag_type fonly =
        std::regex_constants::format_first_only;
```

```
    *std::regex_replace(&buf[0], first, last, rx, fmt) = '\0';
    std::cout << &buf[0] << std::endl;

    *std::regex_replace(&buf[0], first, last, rx, fmt, fonly) = '\0';
    std::cout << &buf[0] << std::endl;

    std::string str("adaeaf");
    std::cout << std::regex_replace(str, rx, fmt) << std::endl;

    std::cout << std::regex_replace(str, rx, fmt, fonly) << std::endl;

    return 0;
}
```

第 18 章 用于大型程序的工具

1. 有时，一个单独的 catch 语句不能完整地处理某个异常。在执行了某些校正操作之后，当前的 catch 可能会决定由调用链更上一层的函数接着处理异常。通过一个空的 catch 语句来重新抛出，将异常传递给另外一个 catch 语句。

```
Throw;
```

2. 如果在改变了参数的内容后 catch 语句重新抛出异常，则只有当 catch 异常声明是引用类型时我们对参数所做的改变才会被保留并继续传播：

```
catch(my_error &eObj){          // 引用类型
    eObj.status=errCodes::serverErr;
    throw;                      // 异常对象改变了
}catch(other_error eObj){       // 非引用类型
    eObj.status=errCodes::badErr;
    throw;                      // 这里异常对象并没有改变
}
```

3. 使用 catch(...)一次性捕获所有异常，catch(...)能单独出现，也能与其他几个 catch 语句一起出现：

4. 构造函数在进入函数体之前首先执行初始值列表，此时在构造函数体内的 try 语句块还没生效，所以构造函数体内的 catch 语句无法处理构造函数初始值列表抛出的异常。

   要可以将构造函数写成函数 try 语句块的形式来处理构造函数初始值抛出的异常：

```
Template <typename T>
Blob<T>::Blob(std::initializer_<T>i l) try:
    data(std::make_shared<std::vector<T> (il)){
/* 空函数体 */
}catch(const std::bad_alloc &e) {handle_out_of_memory(e);}
```

注意：关键字 try 出现在表示构造函数初始值列表的冒号以及表示构造函数体的花括号之前。

5. 在 C++11 中，我们可以通过提供 noexcept 说明指定某个函数不会抛出异常：

```
Void recoup(int) noexcept;              // 不会抛出异常
void alloc(int);                        // 可能会抛出异常
```

noexcept 说明要么在该函数所有声明语句和定义语句，要么一次也不出现。

6. 在成员函数中，noexcept 说明需要跟在 const 及引用限定符之后，而在 final、override 或虚函数的=0 之前。

7. noexcept 说明符接受一个可选的实参，该实参必须能转换为 bool 类型：如果实参为 true，则函数不会抛出异常；如果实参是 false，则函数可能抛出异常：

```
Void recoup(int) noexcept(true);
```

8. noexcept 运算符是一个一元运算符，它的返回值是一个 bool 类型的右值常量表达式，用于表示给定的表达式是否会抛出异常。和 sizeof 类型，noexcept 也不会求其运算对象的值。

```
noexcept(recoup(i));                // 如果 recoup 不抛出异常则结果为 true
```

9. 尽量不要使用 using namespace std，取而代之使用 using std:xxx，因为使用 STL 中有部分名称是没有加下划线的保留标记的，而在自己的源代码中用到了后会引发未定义的后果。

10. 常规的查找规则：由内向外依次查找每个外层作用域。这也说明外层变量会被内部变量所覆盖。

11. A::B::f3()即类似如下的结构：

```
Namespace A
{
    class B
    {
        public:
            int f3();
    }
}
```

12. using 声明语句声明的是一个名字，而不是一个函数;

```
Using ns::print(int);                   // 错误，不能指定形参列表
using ns::print;                        // 正确，using 声明只声明一个名字
```

所以使用 using 声明后，print 所有版本都被引入当前作用域中。

13. 如果命令空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被添加到重载集合中。

14. 多重继承如果从多个基类中继承了相同的构造函数（即形参表完全相同），则程序将

产生错误：

```
Struct Base1
{
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);
}
Struct Base2
{
    Base2() = default;
    Base2(const std::string&);
    Base2(int);
}

struct D1:public Base1,public Base2{
    using Base1::Base1;                    // 出错，形参表完全相同
    using Base2::Base2;
}
```

这里要注意，struct 也能继承。

15. 对于结构体的继承，和类的继承类似，同样有构造函数和析构函数。
16. 派生类的析构函数只负责清除派生类本身分配的资源。析构函数的调用顺序正好和构造函数相反。
17. 派生类的作用域嵌套在直接基类和间接基类的作用域中，继承的名字查找是沿着继承体系自底向上进行，直到找到所需的名字。
18. 虚基类的对象的构造顺序与一般的顺序稍有区别：首先使用提供给最低层派生类的构造函数的初始值初始化对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序依次对其进行初始化。
19. 派生类指针与基类指针的隐式转换：在单个基类情况下，派生类的指针或引用可以自动转换为基类的指针或引用，对于多重继承也是如此，派生类的指针或引用可以转换为其任意其类的指针或引用。

```
class A{}；
class B：public A{}
void function（const A &）；

B b；
function(b);
```

调用 fuction(b)之后，默认把 B 类型转换成 A 类型。

20. 如果派生类的继承方式是 public，则派生类指针与基类指针的隐式转换可以在类的外部正常转换；而如果继承方式是 private 或者 protected，则无法进行转换，系统会报错。

```
#include <iostream>
using std::cout;
using std::endl;
```

```
class base
{
public:
	virtual void func()
		{
		   cout << "call base" << endl;
		}
	virtual ~base()
		{
		   cout << "call ~base" << endl;

		}
};

	class child:public base                          // 必须是 public，否则会报错
	{
	public:
		virtual void func()
			{
			   cout << "call child" << endl;
			}
		virtual ~child()
			{
			   cout << "call ~child" << endl;

	}
};

int main() {
	base *p=new child;                               // 自动把派生类指针转换为基类指针
	p->func();
	delete p;
	return 0;
}
```

21. 不管以什么方式继承，派生类的成员函数和友员函数都能使用派生类向基类的转换。派生类向其直接基类的类型转换对于派生类的成员函数和友员函数来说永远是可访问的。

```
class B{}

class D:private B              //这里以 private 或者 protected 或者 public 都可以

{
	void f()

	{
```

```
        B * base=new D;                 //编译正确
    }
}
```

第 19 章 特殊工具与技术

1. 当我们调用 new 表达式时，new 表达式调用一个名为 operator new（或者 operator new[]）的标准库函数。同理，调用 delete 表达式时，系统调用一个名为 operator delete（或 operator delete[]）的标准库函数。

```
String *str = new string("a value");
```

2. 如果应用程序希望控制内存分配的过程，则它们需要定义自己的 operator new 函数和 operator delete 函数。编译器将使用我们自定义的版本替换标准库版本。
3. 应用程序可以在全局作用域中定义 operator new 函数和 operator delete 函数，也可以将它们定义为成员函数。当编译器发现一条 new 表达式或 delete 表达式时，将在程序中查找可供调用的 operator 函数。如果没有找到，则使用标准库定义的版本。
4. 通过运行时类型识别（RTTI），程序能够使用基类的指针或引用来检索这些指针或引用所指对象的实际派生类型。
5. 通过下面两个操作符提供 RTTI：

- typeid 操作符，返回指针或引用所指对象的实际类型。

  typeid 表达式形如：

```
typeid(e)
```

  这里 e 是任意表达式或者是类型名。

- dynamic_cast 操作符，将基类类型的指针或引用安全地转换为派生类型的指针或引用。

```
dynamic_cast <type*>(e)
dynamic_cast <type&>(e)
dynamic_cast <type&&>(e)
```

其中 type 必须是一个类类型，通常该类应该含有虚函数。第一种形式中，e 必须是一个有效的指针，第二种形式中，e 必须是一个左值，在第三种形式中，e 不能是左值。

在上面的所有形式中，e 的类型必须符合以下三个条件之一：e 的类型是目标 type 的公有派生类、e 的类型是目标 type 的仅有基类或者 e 的类型就是目标 type 的类型。如果符合，则类型可以转换成功。否则，转换失败。

举个例子，假设 Base 类至少含有一个虚函数，Derived 是 Base 的公有派生类：

```
if(Derived *dp = dynamic_cast<Derived*>(bp))
{
    /* 语句内容 */
}else{
    /* 语句内容 */
}
```

6. 可以使用 dynamic_cast 操作符将基类类型对象的引用或指针转换为同一继承层次中其他类型的引用或指针。与 dynamic_cast 一起使用的指针必须是有效的，它必须为 0 或者指向一个对象。
7. 与其他强制类型转换不同，dynamic_cast 涉及运行时类型检查。如果绑定到引用或指针的对象不是目标类型的对象，则 dynamic_cast 失败。
8. typeid 操作的返回值是 const type_info_p&，type_info_p 是标准库类型 type_info 或其公有派生类。
9. 通常情况下，我们使用 typeid 比较两条表达式的类型是否相同：

```
Derived *dp = new Derived;
Base *bp = dp;                          // 两个指针都指向 Derived 对象
if(typeid(*bp)==typeid(*dp))
{
    /* 语句内容 */
}
```

只有当类型含有函数时，编译器才会对表达式求值。反之，如果类型不含有虚函数，则 typeid 返回表达式的静态类型，编译器无须对表达式求值也能知道表达式的静态类型。

10. type_info 类至少提供下列操作：

- t1===t2

  如果 type_info 对象 t1 和 t2 表示同一种类型，返回 true。

- t1!=t2

  如果 type_info 对象 t1 和 t2 表示不同类型，返回 true。

- t.name()

  返回一个 C 风格的字符串，表示类型的可打印形式。类型名字的生成方式因系统而异。

- t1.before(t2)：返回一个 bool 值，表示 t1 是否位于 t2 之前。

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct Base {};
struct Derived : Base {};
struct Poly_Base {virtual void Member(){}};
struct Poly_Derived: Poly_Base {};

int main() {

   int a;
   int * pa;
   char *p;
   cout << "int is: " << typeid(p).name() << endl;
   cout << "  a is: " << typeid(a).name() << endl;
```

```
    cout << " pa is: " << typeid(pa).name() << endl;
    cout << "*pa is: " << typeid(*pa).name() << endl << endl;
}
```

11. C++11 新引入了限定作用域的枚举类型，来解决传统的枚举类型作用域不受限制的问题。定义格式类似如下：

```
enum class enum_name{enum_val1,enum_val2,enum_val3};
```

例如：

```
enum class Color { RED, BLACK };
myColor c = Color::RED;
```

12. C++11 新标准中，可以有 enum 的名字后加上冒号以及我们想在该 enum 中使用的类型：

```
enum intValues:unsigned long long{
      charTyp=255,shortTyp=65535
};
```

如果没有指定 enum 的潜在类型，默认情况下限定作用域的 enum 成员类型是 int。

13. 在 C++11 新标准中，可以提前声明 enum。enum 的前置声明必须指定其成员大小：

```
Enum intValues:unsigned long long;                    // 不限作用域的，必须指定成员类型
enum class open_mods;                          // 限定作用域的枚举类型可以使用默认类型 int
```

14. 成员指针是指可以指向类的非静态成员的指针。

15. 与普通指针不同的是，成员指针还必须包含成员所属的类。因此，我们必须在*之前添加 classname::以表示当前定义的指针可以指向 classname 的成员。

```
Const string screen::*pdata;
```

上述语句将 pdata 声明成“一个指向 screen 类的 const string 成员的指针”。

16. 当我们初始化一个成员指针时，需指定它所指的成员。例如，我们可以令 pdata 指向某个非特定 screen 对象的 contents 成员：

```
Const string screen::*pdata;
auto pdata=&screen::contents;
```

17. 当我们初始化一个成员指针或为成员指针赋值时，该指针并没有指向任何数据。成员指针指定了成员而非该成员所属的对象，只能当解引用成员指针时我们才提供对象的信息。

18. 有两种成员指针访问运算符：.*和->*，这两个运算符使得我们可以解引用指针并获得该对象的成员：

```
Const string Screen::*pdata;
auto pdata=&Screen::contents;
Screen myScreen,*pScreen=&myScreen;
// .*解引用 pdata 以获得 myScreen 对象的成员 contents 成员
auto s= myScreen.*pdata;
```

```
// ->*解引用 pdata 以获得 pScreen 所指对象的 contents 成员
s=pScreen->*pdata;
```

19. 常规的访问控制规则对成员指针同样有效。例如，如果 Screen 的 contents 成员是私有的，则对于 pdata 的使用必须位于 Screen 类的成员或友元内部，否则程序将发生错误。

20. 指向成员函数的指针也使用 classname::*形式来声明：

```
Auto pmf = &Screen::get_cursor;
```

- 如果成员存在重载问题，则必须显式地声明函数类型以明确指出想要使用的是哪个函数。例如，我们可以声明一个指针，令其指向含有两个形参的 get：

```
Char (Screen::*pmf2)(Screen::pos,Screen:pos)const;
pmf2=&Screen::get;
```

出于优先级的考虑，Screen::*两端的括号必不可少。

- 和使用指向数据成员的指针一样，我们使用.*或者->*运算符作用于指向成员函数指针，以调用类的成员函数：

```
Screen myScreen,*pScreen=&myScreen;
// 通过 pScreen 所指的对象调用 pmf 所指的函数
char c1=(pScreen->*pmf)();
// 通过 myScreen 对象将实参传参含有两个形参的 get 函数
char c2=(myScreen.*pmf2)(0,0)
```

21. 使用 using 还可以给模板起别名，语法如下：

```
template<class T> struct Alloc { /* ... */ };
template<class T>
using Vec = std::vector<T, Alloc<T>>;
```

22. 使用 using 或 typedef 可以给成员指针的类型起别名，语法如下：

```
Using Action = char (Screen::*) (Screen::pos,Screen::pos) const;
Action get = &Screen::get;
```

Action 是成员指针的类型的别名。

23. 成员指针不是一个可调用对象，这样的指针不支持函数调用运算符：

```
Auto fp=&string::empty;              // fp 指向 string 的 empty 函数
find_if(svec.begin(),svec.end(),fp);        // 错误，必须使用.*或->*调用成员指针
```

- 从指向成员函数的指针获取可调用对象的一种方法是使用标准库模板 function：

```
function<bool (const string&)> fcn=&string::empty;
find_if(svec.begin(),svec.end(),fcn);
```

empty 是一个接受 string 参数并返回 bool 值的函数。

- 也可以使用标准库功能 mem_fn 来让编译器负责推断成员的类型。

```
find_if(svec.begin(),svec.end(),mem_fn(&string::empty));
```

- 还可以使用 bind 从成员函数生成一个可调用对象：

```
Auto it = find_if(svec.begin(),svec.end(),bind(&string::empty,_1);)
```

当我们使用 bind 时，必须将函数中用于表示执行对象的隐式形参转换为显式的。

C 和 C++编程

1. 指针加 1 的情况要结合指针指向的值的类型来判断。对于指向 int 型数据的指针，指针加 1 后编译器处理效果是指针值加 sizeof(int)，也就是加 4，对于 char 型数据的指针，指针加 1 后编译器实际处理是指针加 sizeof(char)，依此类推。

```
#include <iostream>
#include <memory>
#include <cmath>
using  namespace std;

int main(int argc,char *argv[]) {
	int a[]={0,1,2,3,4,5};
	int *i=a;
	i++;
	cout <<*i<<endl;
	cout <<*++i<<endl;
}
```

编译器输出：

```
1
2
```

2. 结构体和共用体 union 的区别在于：结构体的各个成员会占用不同的内存，互相之间没有影响；而共用体的所有成员占用同一段内存，修改一个成员会影响其余所有成员。

```
union TokenValue {
char cval;
int ival;
double dval;
};
```

定义联合的同时创建变量：

```
union data{
	int n;
	char ch;
	double f;
} a, b, c;
```

其中如果不再需要定义变量，联合名可省略：

```
union{
```

```
    int n;
    char ch;
    double f;
} a, b, c;
```

注意：是联合的各个成员都占用同一块内存，而不是变量占用同一块内存，也就是说，上面的 n、ch 和 f 都占用相同的内存。

3. union 中可以定义多个成员，union 的大小由最大的成员的大小决定。
4. 在 C++新标准中，含有构造函数或析构函数的类类型也可以作为 union 的成员类型。union 可以为其成员指定 public、protected 和 private 等保护标记。默认情况下，union 的成员都是公有的。
5. 内存对齐有利于系统 CPU 提高访问速度。
6. struct/class 内存对齐原则：以第一个数据成员的位置的 offset 为 0，以后每个数据成员的存储位置的偏移量必须要是该数据成员大小的整数倍，小于 4 个字节数据除外。如 int 型数据的存储位置偏移量为 4、8 等 4 的倍数。

```
union PageLayout
{
    struct
    {
        int page_index;          // 这里占据 4 个字节
        char key;
        short a;
        char b;              // key、a 和 b 占据 4 个字节
        double d;            // 这里独占 8 个字节
        char c;              // 这里独占 8 个字节
    }m;
    char dummy[10];
}PageLayout;
```

大小共 32 个字节。

7. 根据内存对齐原则，可以有以下结论：

- struct 或 class 的总大小是最大数据成员的整数倍；
- 定义 struct 或 class 数据成员时最好按大小顺序从小到大或从大到小的顺序定义。

8. 函数声明时可以不写形参，只留下类型：

```
char    *sock_str_flag(union val *, int);
```

9. do..while(0)有两个用法：

(1) 用于宏定义：

```
#define FOO(x) do {\
    some_code_line_1;\
    some_code_line_2;\
} while (0)
```

然后调用 FOO():

```
if ( someCond )
    FOO(x)
else
    //...
```

由于 do..while(0)是一个代码块，所以在 if 语句展开后依然是一个代码块。

- 注意这里的关键是 FOO(X)后面不能有分号，分号会终结整个 if 语句，导致后面的 else 找不到匹配对象。下面这样是错的：

```
if ( someCond )
    FOO(x);
else
    //...
```

(2) 消除代码冗余和 goto 语句：

```
bool Execute()
{
    // 分配资源
    int *p = new int;

    bool bOk(true);
    do
    {
        // 执行并进行错误处理
        bOk = func1();
        if(!bOk) break;

        bOk = func2();
        if(!bOk) break;

        bOk = func3();
        if(!bOk) break;

        // .....

    }while(0);

    // 释放资源
    delete p;
    p = NULL;
    return bOk;

}
```

使用 goto 语句也可以实现同样的功能，但不安全。

10. sizeof 总结：

(1) sizeof 是操作符，不是函数。

(2) sizeof 不能求得 void 类型的长度：

```
sizeof(void);
sizeof(fun())
```

这些都是不对的。

(3) sizeof 能求得 void 类型的指针的长度，也就是能求 void *的长度：

```
sizeof(void *)
```

这个是对的。

(4) sizeof 不能求得动态分配的内存的大小。

(5) sizeof 不能对不确定的数组的数组求内存占用大小。

(6) 当表达式作为 sizeof 的操作数时，它返回表达式的计算类型大小，但是它不对表达式求值：

```
char ch=1;
int num=1;
int n1=sizeof(ch+num);          //结果 n1=4
int n2=sizeof(ch=ch+num);    //n2=1，ch=1
```

(7) sizeof 可以对函数调用求大小，并且求得的大小等于函数返回类型的大小，但是不执行函数体。

(8) sizeof 求得结构体（及其对象）的大小并不等于各个数据成员对象的大小之和。这个跟结构体内存对齐有关。

(9) sizeof 不能用于求结构体的位域成员的大小，但是可以求得包含位域成员的结构体的大小。

11. 引用是变量的别名，而指针是变量的地址。（引用、别名）

12. 数组：

(1) 数组的维数必须在一对方括号 [] 内指定，方括号内必须是常量或 const 变量，不能是普通变量：

```
int staff_size = 27;
char input_buffer[buf_size];
double salaries[staff_size];                    //这个定义会出错，因为[]内是变量
```

(2) 显式初始化的数组不需要指定数组的维数值，编译器会根据列出的元素个数来确定数组的长度：

```
int ia[] = {0, 1, 2};
```

(3) 如果维数大于列出的元素初值个数，则只初始化前面的数组元素；剩下的其他元素，若是内置类型则初始化为 0，若是类类型则调用该类的默认构造函数进行初始化：

int a[3]={1}; //第一个元素为 1，其余初始化为 0

13. C++ 提供了一种特殊的指针类型 void*，它可以保存任何类型对象的地址。

14. *ptr 表示 ptr 的值本身是一个指针，而*ptr 的值就是指针指向的值。

15. 在 C 语言中，函数名就是首地址。

16. 函数指针变量，类似于变量，可以对其进行赋值：

```
#include <stdio.h>
#include <stdlib.h>

void (*funP)(int);  //声明也可写成 void(*funP)(int x)，但习惯上一般不这样。
void (*funA)(int);
void myFun(int x);   //声明也可写成：void myFun( int );
int main()
{
    myFun(100);

    funP=&myFun;              //将 myFun 函数的地址赋给 funP 变量
    (*funP)(200);             //通过函数指针变量来调用函数

    //myFun 与 funA 的类型关系类似于 int 与 int 的关系。
    funA=myFun;
    funA(300);

    return 0;
}

void myFun(int x)
{
    printf("myFun: %d\n",x);
}
```

声明 void (*funA)(int);表示 fun 是一个指针，指向一个函数。

17. 指针函数和函数指针：

(1) 指针函数：返回值是一个地址，指针函数不带括号：

```
int *GetDate();
int * aaa(int,int);                 // 调用时和普通函数一样使用，例如 aaa(5,2)
```

(2) 函数指针：返回值是一个指针，指向函数地址，可用它来调用函数，函数指针带括号：

```
void (*fptr)();
fptr=&Function;
```

18. 分析类似下面这类复杂表达式时，需要结合优先级和结合性：

```
int (*(ff(int)))(int *, int);              // 用括号将 ff(int)再括起来也就意味着，ff 是一个函数。
```

根据优先级和结合性得到，*(ff(int))是一个指针函数，该函数返回一个指针。然后再跟后面的(int *,int)结合，得到该指针指向一个另函数。

19. C++的指针管理：

(1) 写一个 new 语句时，立即把 delete 语句也写了。

(2) 函数跳转时，要时刻注意需不需要写上 delete 语句。

20. 智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。

21. 智能指针本质上还是指针，只是会自动释放内存而已。

22. 四种常用智能指针：

(1) 初级智能指针 auto_ptr：思路是将基本类型指针封装为类对象指针，并在析构函数里编写 delete 语句，删除指针指向的内存空间。当指针 auto_ptr 占据的内存被释放后，auto_ptr 指向的内存也自动释放。

- 定义 auto_ptr：

```
auto_ptr<int> pi = new int(1024)
或
auto_ptr<int> pi(new int(1024))
```

定义一个智能指针，指向一个智能指针 pi，指针指向的值为 1024。

- auto_ptr 相关操作：

auto_ptr<T> ap1(ap2);	创建名为 ap1，并保存原来存储在 ap2 中的指针。将所有权转给 ap1，ap2 成为未绑定的 auto_ptr 对象。
ap1 = ap2	删除 ap1 指向的对象，并且使 ap1 指向 ap2 指向的对象，ap2 成为未绑定状态。
~ap	析构函数。删除 ap 指向的对象
*ap	返回对 ap 所绑定的对象的引用
ap.reset(p)	如果 p 与 ap 的值不同，则删除 ap 指向的对象，并且将 ap 绑定到 p
ap.release()	返回 ap 所保存的指针并且使 ap 成为未绑定的
ap.get()	返回 ap 保存的指针
ap->	返回 ap 保存的指针

- auto_ptr 只能用于管理从 new 返回的一个对象，它不能管理动态分配的数组。
- 复制（或者赋值）普通指针是复制（或者赋值）地址，在复制（或者赋值）之后，两个指针指向同一对象。在复制（或者赋值）auto_ptrs 对象之后，原来的 auto_ptr 对象成为未绑定状态，而新的 auto_ptr（左边的 auto_ptr 对象）拥有基础对象。
- auto_ptr 智能指针所指的对象在超出作用域时自动删除。如果发生异常，则对象也超出作用域，析构函数将自动运行的析构函数作为异常处理的一部分。

➢ 在 C++11 中已废弃，仅用于向后兼容。

(2) shared_ptr 是一个引用计数智能指针，这种智能指针隐含复制和赋值运算的所有相关对象共享同一个引用计数。

```
class Example
{
public:
	Example() : e(1) { cout << "Example Constructor..." << endl; }
	~Example() { cout << "Example Destructor..." << endl; }

	int e;
};

int main() {

	shared_ptr<Example> pInt(new Example());
	cout << (*pInt).e << endl;
	cout << "pInt 引用计数: " << pInt.use_count() << endl;

	shared_ptr<Example> pInt2 = pInt;
	shared_ptr<Example> pInt3(new Example());
	cout << "pInt 引用计数: " << pInt.use_count() << endl;
	cout << "pInt2 引用计数: " << pInt2.use_count() << endl;
	cout << "pInt3 引用计数: " << pInt3.use_count() << endl;
}
```

输出为：

```
pInt 引用计数:1
pInt 引用计数: 2
pInt2 引用计数: 2
pInt3 引用计数:1
```

例子说明：

- A 对象使用 B 对象来初始化时，B 对象引用计数加 1，并和 A 对象共享同一个引用计数；
- 新建 shared_ptr 指针只要不涉及现有 shared_ptr 指针，则引用计数为 1，也就是只有该指针自己。

- make_shared()：该函数会在堆中分配一个对象并初始化，最后返回指向此对象的 share_ptr 实例。

```
shared_ptr<string> pStr = make_shared<string>(10, 'a');
```

传递给 make_shared 函数的参数必须和 shared_ptr 所指向类型的某个构造函数相匹配。

- use_count()：查看资源的引用计数。

  reset()：重置指针。

  get()：获取指针值。

  unique()：检查是否是唯一值。

swap()：交换两个 shared_ptr 指针的值。

- shared_ptr 的使用方式与普通指针的使用方式类似，既可以使用解引用操作符*获得原始对象进而访问其各个成员，也可以使用指针访问符->来访问原始对象的各个成员。
- 不要在函数实参中创建 shared_ptr。因为 C++的函数参数的计算顺序在不同的编译器下是不同的。正确的做法是先创建好，然后再传入。

(3) 如果说两个 shared_ptr 相互引用，那么这两个指针的引用计数永远不可能下降为 0,资源永远不会释放。weak_ptr 是用来解决 shared_ptr 相互引用时的死锁问题的。它的最大作用在于协助 shared_ptr 工作，像旁观者那样观测资源的使用情况。

- weak_ptr 没有共享资源，它的构造不会引起指针引用计数的增加。
- lock()：锁定并恢复 weak_ptr，即获得一个可用的 shared_ptr 对象，从而操作资源。

  expired()：检查是否过期。
- weak_ptr 有 3 个和 shared_ptr 作用类似的函数：reset()、swap()、use_count()。

(4) unique_ptr 是用于取代 c++98 的 auto_ptr 的产物，是一个独享所有权的智能指针，它提供了严格意义上的所有权，包括：

- ◆ 拥有它指向的对象
- ◆ 无法进行复制构造，无法进行复制赋值操作。即无法使两个 unique_ptr 指向同一个对象。但是可以进行移动构造和移动赋值操作。
- ◆ 保存指向某个对象的指针，当它本身被删除释放的时候，会使用给定的删除器释放它指向的对象

- unique_ptr 对象删除它们的管理对象而不考虑是否其他指针仍然指向相同的对象，因此可能会留下任何其他指向指向无效位置的指针。
- unique_ptr 和 auto_ptr 用法很相似，不过不能使用两个智能指针赋值操作，应该使用 std::move；而且它可以直接用 if(xxx == NULL)来判断是否空指针。
- 相关函数：

  get()：获取指针值。

  get_deleter()：获取删除器。

  release()：释放指针。

  reset()：重置指针。

  swap()：交换两个指针的值。

23. 关键字 auto 用于类型推导，也就是自动判断变量的类型。类型推导发生在编译期：

(1) auto 的合适用于代替冗长复杂、变量使用范围专一的变量声明

```
std::vector<std::string> vs;
    for (std::vector<std::string>::iterator i = vs.begin(); i != vs.end(); i++)
    {
        //...
    }
```

使用 auto 代替如下：

```
std::vector<std::string> vs;
    for (auto i = vs.begin(); i != vs.end(); i++)
    {
        //..
    }
```

(2) auto 变量必须在定义时初始化。

24. 注意 struct 结构体的这种用法：

```
struct sock_opts {
 const char        *opt_str;
 int               opt_level;
 int               opt_name;
 char   *(*opt_val_str)(union val *, int);
} sock_opts[] = {
 { "SO_BROADCAST",      SOL_SOCKET,      SO_BROADCAST,   sock_str_flag },
 { "SO_DEBUG",          SOL_SOCKET,      SO_DEBUG,       sock_str_flag },
}
```

这是直接给 socket_opts 数组赋值。

25. return 语句有两种写法：

```
Return 1;
return(1);
```

26. Const 作用

(1) 定义 const 常量　　（const、常量）

```
const int Max = 100;
```

(2) 便于进行类型检查　（类型检查）

```
void f(const int i) { .....}
    //对传入的参数进行类型检查，不匹配进行提示
```

(3) 可以保护被修饰的东西　　（保护、被修饰）

```
void f(const int i) { i=10;//error! }
    //如果在函数体内修改了 i，编译器就会报错
```

(4) 为函数重载提供了一个参考　　（重载）

```
class A
{
            ......
  void f(int i)     {.....} //一个函数
  void f(int i) const {.....} //上一个函数的重载
```

```
        ......
};
```

(5) 可以节省空间，避免不必要的内存分配　　（节省空间）

```
#define PI 3.14159        //常量宏
const doulbe  Pi=3.14159; //此时并未将 Pi 放入 ROM 中
                  ......
double i=Pi;   //此时为 Pi 分配内存，以后不再分配！
double I=PI;  //编译期间进行宏替换，分配内存
double j=Pi;  //没有内存分配
double J=PI;  //再进行宏替换，又一次分配内存！
```

27. Const 的使用

(1) 定义常量

(2) 指针使用 CONST

- 指针本身是常量不可变

```
char* const pContent;
```

- 指针所指向的内容是常量不可变

```
const char *pContent;
```

- 两者都不可变

```
const char* const pContent;
```

如果 const 位于*的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量；如果 const 位于*的右侧，const 就是修饰指针本身，即指针本身是常量。（const、左侧、指向内容不可变、右侧、指针不可变）

28. chmod 函数和 chmod 命令并不一样，chmod 函数第二个参数不是数字，而是一些宏。
例如：（chmod 函数、第二个参数、宏）

- S_ISUID：文件执行时，内核将其进程的有效用户组 ID 设置为文件的所有者 ID
- S_ISGID：文件执行时，内核将其进程的有效用户组 ID 设置为文件的所有组 ID
- S_IRUSR：文件所有者具可读取权限
- S_IWUSR：文件所有者具可写入权限
- S_IXUSR：文件所有者具可执行权限
- S_IRGRP：用户组具可读取权限
- S_IWGRP：用户组具可写入权限
- S_IXGRP：用户组具可执行权限
- S_IROTH：其他用户具可读取权限

- S_IWOTH：其他用户具可写入权限
- S_IXOTH：其他用户具可执行权限

```
#include "apue.h"

int
main(void)
{
        struct stat             statbuf;

        /* turn on set-group-ID and turn off group-execute */

        if (stat("foo", &statbuf) < 0)
                err_sys("stat error for foo");
        if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
                err_sys("chmod error for foo");

        /* set absolute mode to "rw-r--r--" */

        if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
                err_sys("chmod error for bar");

        exit(0);
}
```

29. typedef有四种用法：　　（别名、结构体名称、无关类型、复杂声明）

- 自定义变量别名

```
typedef char* PCHAR;            // 一般用大写
PCHAR pa, pb;
```

- 自定义结构体名称

```
typedef struct {
        float real;
        float imag;
}COMPLEX;

COMPLEX a;              // 定义结构体变量
```

- 定义与平台无关的类型

比如定义一个叫 REAL 的浮点类型，在目标平台一上，让它表示最高精度的类型为：

```
typedef long double REAL;
```

在不支持 long double 的平台二上，改为：

```
typedef double REAL;
```

在连 double 都不支持的平台三上，改为：

```
typedef float REAL;
```

- 为复杂的声明定义一个新的简单的别名 （复杂声明、简单的别名）

对复杂变量的声明，只要记住在传统声明定义表达式里用类型名代替变量名，然后在开头加上 typedef。这种类型的声明可以用于简化指针函数等：

```
typedef int *(*pFun)(int, char*);
pFun a[5];
```

30. st_atim 指的是 access time，即文件访问时间，而 st_mtim 指的是 modification time，即文件的修改时间。（atim、access time、mtim、modification time）

UNIX 网络编程卷一

第 1 章 简介

1. 要编写通过计算机通信的程序，首先要确定这些程序相互通信所用的协议。（首先确定、协议）
2. 任何现实世界的程序都必须检查每个函数调用是否返回错误。（必须、检查函数、是否返回错误）
3. 只要一个 Unix 函数中有错误发生，全局变量 errno 就被设置为一个指明该错误类型的正值，函数本身则通常返回-1。（错误、errno 被设置）
4. 网络编程中常用到 bind 函数，需要绑定 IP 地址，这时可以设置 INADDR_ANY。INADDR_ANY 就是表示本机的所有 IP，因为有些机子不止一块网卡，多网卡的情况下，这个就表示所有网卡 ip 地址的意思。（INADDR_ANY、多网卡、所有网卡 IP）

```
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(13);    /* daytime server */

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);
```

5. bind()用于绑定套接字和 IPv4 或 IPv6 结构体。（bind、绑定、套接字和 IP）
6. 通过 time()函数来获得日历时间，如果参数为空（NULL），函数将只通过返回值返回现在的日历时间：（time、日历时间）

```
ticks = time(NULL);
```

7. socket、bind 和 listen 这 3 个调用步骤是任何 TCP 服务器准备所谓的监听描述符的正常步骤。（socket、bind 和 listen、正常步骤）

第 2 章 传输层：TCP、UDP 和 SCTP

1. 环回接口称为 lo，以太网接口称为 eth0。（lo、环回、eth0、以太网）
2. UDP 不保证 UDP 数据报会到达其最终目的地，不保证各个数据报的先后顺序跨网络后保持不变，也不保证每个数据报只到达一次。 （UDP、不保证）
3. UDP 是无连接的服务，UDP 客户和服务器之间不必存在任何长期的关系。（UDP、无连接）
4. TCP 告知对端在任何时刻它一次能够从对端接收多少字节的数据，这称为通告窗口。（TCP、告知、数据量、通告窗口）
5. 并发服务器中主服务器循环通过派生一个子进程来处理每个新的连接。（并发、派生子进程）
6. 每个 TCP 套接字有一个发送缓冲区，我们可以使用 SO_SNDBUF 套接字选项来更改该缓冲区的大小。（TCP、发送缓冲区、SO_SNDBUF）

第 3 章 套接字编程简介

1. sockaddr 是一个通用的套接字地址结构：（sockaddr、通用、套接字）

```
Struct sockaddr{
      uint8_t sa_len;
      sa_family_t sa_family;                    // 地址族
      char sa_data[14]                          // 协议地址
}
```

bind()的第二个参数就是一个 sockaddr 结构的地址，所以需要将 IPv4 或 IPv6 地址结构进行强制类型转换。（bind()、sockaddr 类型参数、强制类型转换）

```
bind(sockfd,(struct sockaddr *)&serv,sizeof(serv));
```

2. IPv6 的地址族是 AF_INET6，而 IPv4 的地址族是 AF_INET。
   （IPv6、AF_INET6、IPv4、AF_INET）
3. IPv6 因特网域（AF_INET6)套接字地址用结构 sockaddr_in6 表示：
   （IPv6、sockaddr_in6）

```
struct_in6_addr {
      uint8_t s6_addr[16];
};

struct sockaddr_in6{
      uint8_t sin6_len;
```

```
    sa_family_t sin6_family;   /* 地址族  */
    in_port_t sin6_port;      /* 端口号  */
    uint32_t sin6_flowinfo;
    struct in6—addr sin6_addr;   /* IPv6 地址  */
    uint32_t sin6_scope_id;
};
```

如果系统支持套接字地址结构中的长度字段，那么 sin6_len 常值必须定义。

4. Struct sockaddr_storage 是作为 IPv6 套接字 API 的一部分而定义的新的通用套接字地址结构，它克服了现有的 sockaddr 的一些缺点：（IPv6、通用套接字地址、sockaddr_storage）

```
Struct sockaddr_storage{
    uint8_t ss_len;
    sa_family_t ss_family;
}
```

sockaddr_storage 与 sockaddr 存在两点差别：

- 如果系统支持的任何套接字地址结构有对齐需要，那么 sockaddr_storage 能够满足最苛刻的对齐要求。（满足对齐需要）
- sockaddr_storage 足够大，能够容纳系统支持的任何套接字地址结构。（足够大）

5. 本书共遇到 5 种套接字地址：IPv4、IPv6、Unix 域、数据链和存储。 IPv4 和 IPv6 套接字地址结构是固定长度的，而 Unix 域结构和数据链路结构长度是可变的。为了处理长度可变的结构，当我们把指向某个套接字地址结构的指针作为一个参数传递给某个套接字函数时，也把该结构的长度作为另一个参数传递给这个函数。（五种套接字、IPv4、IPv6、Unix 域、数据链、存储）
6. 当往一个套接字函数传递一个套接字地址结构时，该结构总是以引用形式来传递。该结构的长度也作为一个参数来传递，不过其传递方式取决于该结构的传递方向：是从进程到内核，还是从内核到进程。（传递套接字结构、引用形式）

(1) 从进程到内核传递套接字地址结构的函数有 3 个：bind、connect 和 sendto。这些函数的一个参数是指向某个套接字地址结构的指针，另一个参数是该结构的整数大小，例如：

```
struct sockaddr_in serv;
connect(sockfd, (SA *) &serv, sizeof(serv));
```

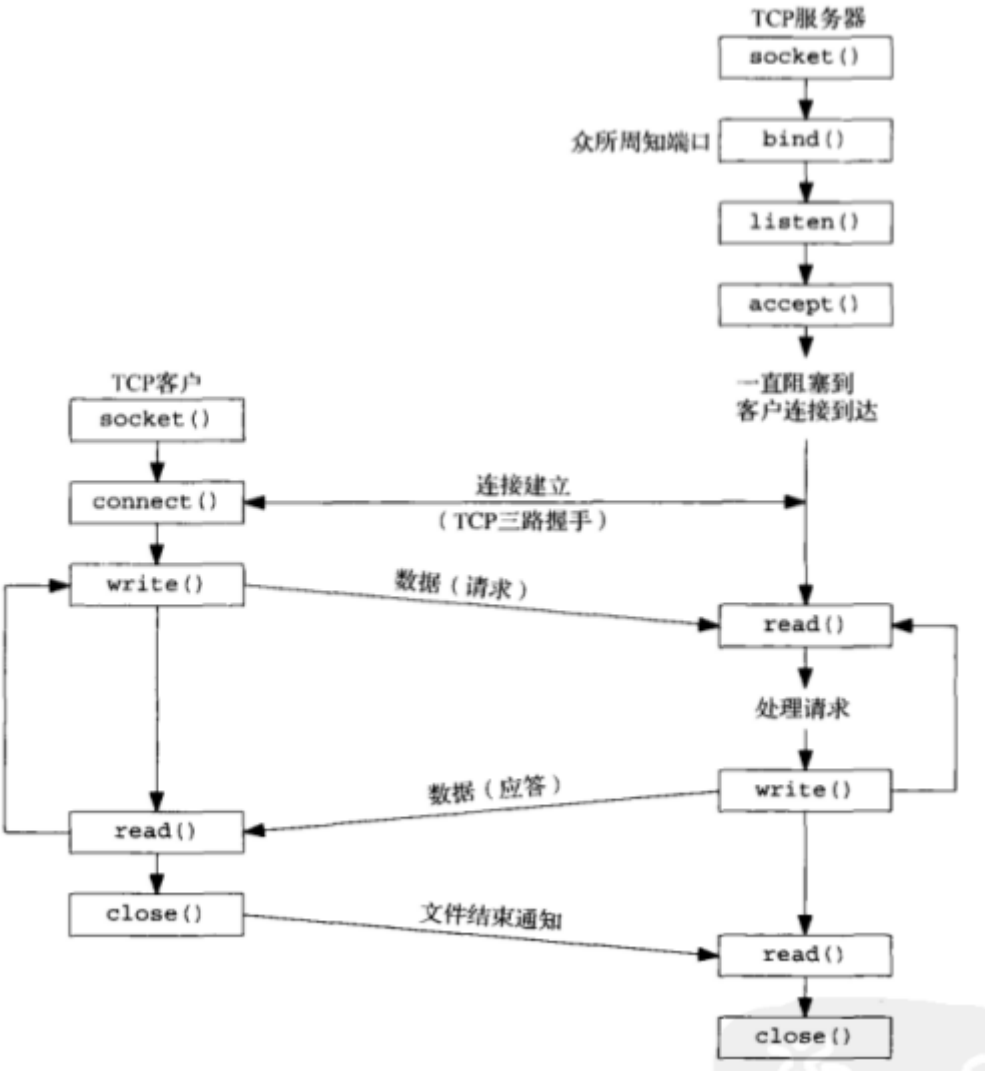
(2) 从内核到进程传递套接字地址结构的函数有 4 个：accept、recvfrom、getsockname 和 getpeername。这 4 个函数的其中两个参数是指向某个套接字地址结构的抱针和指向表示该结体大小的整数变量的指针。

7. 某个给定系统所用的字节序称为主机字节序。（系统所用、主机字节序）
8. 主机字节序和网络字节序之间的转换使用以下 4 个函数：htons()、htonl()、ntohs()和 ntohl()。h 代表 host，n 代表 network，s 代表 short，l 代表 long。如今应该把 s 视为一个 16 位的值（例如端口号），把 l 视为一个 32 位的值（例如 IPv4 地址）。（s、端口、l、IPv4 地址）

9. bzero()把目标字节串中指定数目的字节置为 0。bcopy()将指定数目的字节从源字节串移到目标字节串。bcmp 比较两个任意的字节串，若相同则返回值为 0，否则返回值为非 0。这 3 个函数源于 4.2BSD。
10. memset 把目标字节串指定数目的字节置为值 c。memcpy 类似 bcopy，不过两个指针参数的顺序是相反的。当源字节串与目标字节串重叠时，bcopy 能够正确处理，但是 memcpy 的操作结果却不可知。这种情形下必须改用 ANSIC 的 memmove 函数。

第 4 章 基本 TCP 套接字编程

1. 基本 TCP 客户/服务器程序的套接字函数：



2. AF_前缀表示地址族，PF_前缀表示协议族。（AF_地址族、PF_协议族）
3. 并发服务器中父进程关闭已连接的套接字只是导致相应描述符的引用数值减 1，这个 close 调用并不引发 TCP 的四分组连接终止序列。（关闭、引用数值减 1）
4. 6 个 exec 函数之间的区别在于：

(1) 待执行的程序文件是由文件名（filename)还是由路径名（pathname)指定；（文件名、路径名）

(2) 新程序的参数是一一列出还是由一个指针数组来引用；（列出、数组来引用）

(3) 把调用进程的环境传递给新程序还是给新程序指定新的环境。

5. socket()函数返回的是套接字描述符，而 accept()返回的是代表 TCP 连接的描述符。（socket()、套接字描述符、accept()、连接描述符）

6. fork()子进程的数据是父进程数据的拷贝，而且在父进程中打开的文件，在子进程中同样被打开。（fork()、数据拷贝、同样被打开）

7. 并发服务器的子进程需要关闭套接字描述符，典型的并发服务器程序轮廓：（子进程、关闭套接字描述符）

```
pid_t pid;
int listenfd, connfd;
liscenfd = socket(...);
bind(listenfd,…);
Listen(liscenfd, LISTENQ);
for ( ; ; ) {
    connfd = Accept(listenfd,…);
    if ((pid = fork())== 0)
        close(listeafd);            // 子进程关闭套接字描述符
        doit(connfd);
        close(connfd);
        exit(0);
    }
    close(connfd);
}
```

当一个连接建立时，accept 返回，服务器接着调用 fork，然后由子进程服务客户（通过已连接套接字 connfd)，父进程则等待另一个连接（通过监听套接字 listenfd）。既然新的客户由子进程提供服务，父进程就关闭已连接套接字。

子进程关闭套接字描述符，父进程关闭已连接套接字。（子进程、套接字描述符、父进程、已连接套接字）

8. getsockname()用于获取绑定到一个套接字的地址。getpeername()用于找到对方的地址。（getsockname()、获取套接字地址、getpeername()、找对方地址）

- 在一个没有调用 bind 的 TCP 客户上，connect 成功返回后，getsockname 用于返回由内核赋予该连接的本地 IP 地址和本地端口号。（没有 bind、getsockname、本地 IP 和端口）
- 在以端口号 0 调用 bind (即由内核自己去选择本地端口号）后，getsockname 用于返回由内核赋予的本地端口号。（端口 0、内核选择端口）
- getsockname 可用于获取某个套接字的地址族：

```
Int sockfd_to_family(int sockfd)
{
        struct sockaddr_storage ss;
        socklen_t       len;
```

```
        len = sizeof(ss);
        if (getsockname(sockfd, (SA *) &ss, &len) < 0)
                return(-1);
        return(ss.ss_family);
}
```

- 在一个以通配 IP 地址调用 bind 的 TCP 服务器上，与某个客户的连接一旦建立(accept 成功返回），getsockname 就可以用于返回由内核赋予该连接的本地 IP 地址。（getsockname、返回本地 IP）
- 当一个服务器是由调用过 accept 的某个进程通过调用 exec 执行程序时，它能够获取客户身份的唯一途径便是调用 getpeername。（调用 exec、获取客户身份、getpeername）

第 5 章 TCP 客户/服务器程序示例

1. 信号就是告知某个进程发生了某个事件的通知，有时也称为软件中断。（信号、软件中断）

第 6 章 I/O 复用：select 和 poll 函数

1. 计算机科学中，同步是指两个不同但有联系的概念：进程同步与数据同步。（同步、进程同步、数据同步）
- 进程同步指多个进程在特定点会合或者握手使得达成协议或者使得操作序列有序。（进程同步、多个进程、会合、使操作序列有序）
- 数据同步指一个数据集的多份拷贝一致以维护完整性。常用进程同步原语实现数据同步。（数据同步、数据一致、维护完整性）

2. I/O 模型：

(1) 同步阻塞 IO：一直阻塞，直到有数据准备好。（同步阻塞、一直阻塞）

(2) 同步非阻塞 IO：通过轮询方式，每隔一段时间查询是否有数据准备好。（同步非阻塞、轮询）

(3) IO 多路复用：也就是 poll、select 的方式。（IO 多路复用、poll、select）

(4) 信号驱动式 IO：当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。（信号驱动、准备好时、收到信号）

(5) 异步非阻塞 IO：用户进程发起 aio_read 操作之后，立刻就可以开始去做其它的事。kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会给用户进程发送一个 signal 或执行一个基于线程的回调函数来完成这次 IO 处理过程，告诉它 read 操作完成了。（异步非阻塞、发起操作后、做其他事情）

3. 终止网络连接的通常方法是调用 close 函数。不过 close 有两个限制，却可以使用 shutdown 函数来避免：

(1) close 把描述符的引用计数减 1，仅在该计数变为 0 时才关闭套接字。使用 shutdown 可以不管引用计数就激发 TCP 的正常连接终止序列。（shutdown、不管引用计数、终止连接）

(2) close 终止读和写两个方向的数据传送。shutdown()可以只终止其中一个方向的数据传输。（close 终止、读和写、shutdown()、一个方向）

4. 当一个服务器在处理多个客户时，它绝对不能阻塞于只与单个客户相关的某个函数调用。否则可能导致服务器被挂起，拒绝为所有其他客户提供服务。这就是所谓的拒绝服务型攻击。它就是针对服务器做些动作，导致服务器不再能为其他合法客户提供服务。（不能阻塞、单个客户、某个函数）

第 7 章 套接字选项

1. 有很多方法来获取和设置影响套接字的选项:

(1) getsockopt 和 setsockopt 函数；

(2) fcntl 函数；

(3) ioctl 函数。

2. getsockopt 和 setsockopt 函数的第二个参数是 level，level 指定控制套接字的层次，可以取三种值：（getsockopt、level 参数）

(1) SOL_SOCKET：通用套接字选项

(2) IPPROTO_IP：IP 选项

(3) IPPROTO_TCP：TCP 选项

3. 套接字选项分为两大类：

- 一是启用或禁止某个特性的二元选项，称为标志选项；
- 二是取得并返回我们可以设置或检查的特定值的选项，称为值选项。

4. fcntl 函数提供了与网络编程相关的如下特性：

(1) 非阻塞式 IO。通过使用 F_SETFL 命令设置 O_NONBLOCK 文件状态标志，我们可以把一个套接字设置为非阻塞型。

(2) 信号驱动 IO。通过使用 F_SETFL 命令设置 O_ASYNC 文件状态标志，我们可以把一个套接字设置成一旦其状态发生变化，内核就产生一个 SIGIO 信号。

(3) F_SETOWN 命令允许我们指定用于接收 SIGIO 和 SIGURG 信号的套接字属主。

5. 使用 fcntl 开启非阻塞式 I/O 的典型代码：（fcntl、开启非阻塞式 I/O）

```
int flags;

if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
      err_sys ("F_GETFL error");
flags | = O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
```

```
    err_sys("F_SETFL error");
```

关闭非阻塞标志代码：

```
flags & = ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("F_SETFL error");
```

6. MSS 与 MTU：

- MSS 就是 TCP 数据包每次能够传输的最大数据分段。为了达到最佳的传输效能，TCP 协议在建立连接的时候通常要协商双方的 MSS 值。
- MTU 是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。
- MSS 应该就是 MTU 减去包头。

7. 如果两台主机之间的通信要通过多个网络，那么每个网络的链路层就可能有不同的 MTU。两台通信主机路径中的最小 MTU，被称作路径 MTU。
8. 路径 MTU 发现也就是动态检测路径 MTU 大小。
9. 通用套接字选项：

(1) SO_BROADCAST：本选项开启或禁止进程发送广播消息的能力。

(2) SO_DEBUG：本选项仅由 TCP 支持。当给一个 TCP 套接字开启本选项，内核将为 TCP 在该套接字和接收的所有分组保留详细跟踪信息。这些信息保存在内核的某个环形缓冲区中，并可以使用 trpt 程序进行检查；

(3) SO_DONTROUTE：本选项规定外出的分组将绕过底层协议的正常路由机制，以强制将分组从特定接口送出。

(4) SO_ERROR：获取 errno 的值。该选项只能获取不能设置。

(5) SO_KEEPALIVE：开启保活探测功能。设置该选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方 发一个保持存活探测分节。

(6) SO_LINGER：指定函数 close 对面向连接的协议如何操作（如 TCP）。内核缺省 close 操作是立即返回，如果有数据残留在套接口缓冲区中则系统将试着将这些数据发送给对方。

(7) SO_OOBINLINE：开启本选项时，带外数据将被留在正常的输入队列中。这种情况下接收函数的 MSG_OOB 标志不能用来读带外数据。

(8) SO_RCVBUF 和 SO_SNDBUF：这两个选项可以改变接收缓冲区和发送缓冲区的大小。

(9) SO_RCVLOWAT 和 SO_SNDLOWAT：每个套接字还有一个接收低水位标记和一个发送低水位标记。他们由 select 函数使用，接收低水位标记是让 select 返回可读时套接字接收缓冲区中所需的数据量；发送低水位标记是让 select 返回可写时套接字发送缓冲区中所需的可用空间。这两个套接字选项允许我们修改这两个低水位标记。

(10) SO_RCVTIMEO 和 SO_SNDTIMEO：这两个选项允许我们给套接字的接收和发送设置一个超时值。

(11) SO_REUSEADDR：该选项通知内核，如果端口忙，但 TCP 状态位于

TIME_WAIT，可以重用端口。如果端口忙，而 TCP 状态位于其他状态，重用端口时依旧得到一个错误信息，指明"地址已经使用中"。如果你的服务程序停止后想立即重启，而新套接字依旧使用同一端口，此时 SO_REUSEADDR 选项非常有用。

(12) SO_TYPE：返回套接字类型，返回的整数值是一个诸如 SOCK_STREAM 或 SOCK_DGRAM 之类的值。

10. IPv4 选项：

(1) IP_HDRINCL：如果本选项是给一个原始 IP 套接字设置的，那么我们必须为所有在该原始套接字上发送的数据报构造自己的 IP 首部。

(2) IP_OPTIONS：本选项的设置允许我们在 IPv4 首部中设置 IP 选项。

(3) IP_RECVDSTADDR：本套接字选项导致所收到 UDP 数据报的目的 IP 地址由 recvmsg 函数作为辅助数据返回。

(4) IP_RECVIF：本套接字选项导致所收到 UDP 数据报的接收接口索引由 recvmsg 函数作为辅助数据返回。

(5) IP_TOS：本套接字选项允许我们为 TCP、UDP 套接字设置 IP 首部中的服务类型字段。

(6) IP_TTL：本选项设置或获取系统用在从某个给定套接字发送的单播分组上的默认 TTL 值。

11. IPv6 套接字选项：

(1) IPV6_CHECKSUM：本选项指定用户数据中检验和所处位置的字节偏移。

(2) IPV6_DONTFRAG：开启本选项将禁止为 UDP 套接字或原始套接字自动插入分片首部，外出分组中大小超过发送接口 MTU 的那些分组将被丢弃。应用进程应该开启 IPV6_RECVPATHMTU 选项以获悉路径 MTU 的变动。

(3) IPV6_NEXTHOP：本选项将外出数据报的下一跳地址指定为一个套接字地址结构。

(4) IPV6_PATHMTU：该选项只能获取，返回值为由路径 MTU 发现功能确定的当前 MTU。

(5) IPV6_RECVSTOPTS：开启本选项表明，任何接收到的 IPv6 目的地选项都将由 recvmsg 作为辅助数据返回。默认为关闭。

(6) IPV6_RECVHOPLIMIT：开启本选项表明，任何接收到的跳限字段都将由 recvmsg 作为辅助数据返回。默认为关闭。

(7) IPV6_RECVHOPOPTS：开启本选项表明，任何接收到的 IPv6 步跳选项都将由 recvmsg 作为辅助数据返回。本选项默认为关闭。

(8) IPV6_RECVPATHMTU：开启本选项表明，某条路径的路径 MTU 在发生变化时将由 recvmsg 作为辅助数据返回（不伴随任何数据）。

(9) IPV6_RECVPKTINFO：开启本选项表明，接收到的 IPv6 数据报的以下两条信息将由 recvmsg 作为辅助数据返回：目的 IPv6 地址和到达接口索引。

(10) IPV6_RECVRTHDR：开启本选项表明，接收到的 IPv6 路由首部将由 recvmsg 作为辅助数据返回。本选项默认为关闭。

(11) IPV6_RECVTCLASS：开启本选项表明，接收到的流通类别（包含 DSCP 和

ECN 字段）将由 recvmsg 作为辅助数据返回。本选项默认为关闭。

(12) IPV6_UNICAST_HOPS：本 IPv6 选项类似于 IPV4 的 IP_TTL 套接字选项。设置本选项会给在相应套接字上发送的外出数据报指定默认跳限，获取本选项会返回内核用于相应套接字的跳限值。

(13) IPV6_USE_MIN_MTU：把本选项设置为 1 表明，路径 MTU 发现功能不必执行，为避免分片，分组就使用 IPv6 的最小 MTU 发送。把本选项设为 0 表明，路径 MTU 发现功能对于所有目的地都得执行。把本选项设为-1 表明，路径 MTU 发现功能仅对单播目的地执行，对于多播目的地就使用最小 MTU。默认为 1。

(14) IPV6_V6ONLY：在一个 AF_INET6 套接字上开启本选项将限制它只执行 IPv6 通信。本选项默认为关闭。

(15) ipv6_xxx：大多数用于修改协议首部的 IPv6 选项假设：就 UDP 套接字而言，信息由 recvmsg 和 sendmsg 作为辅助数据在内核和应用进程之间传递；就 TCP 套接字而言，同样的信息改用 getsockopt 和 setsockopt 获取和设置。套接字选项和辅助数据的类型一致，并且访问套接字选项的缓冲区所含的信息和辅助数据中存放的信息也一致。

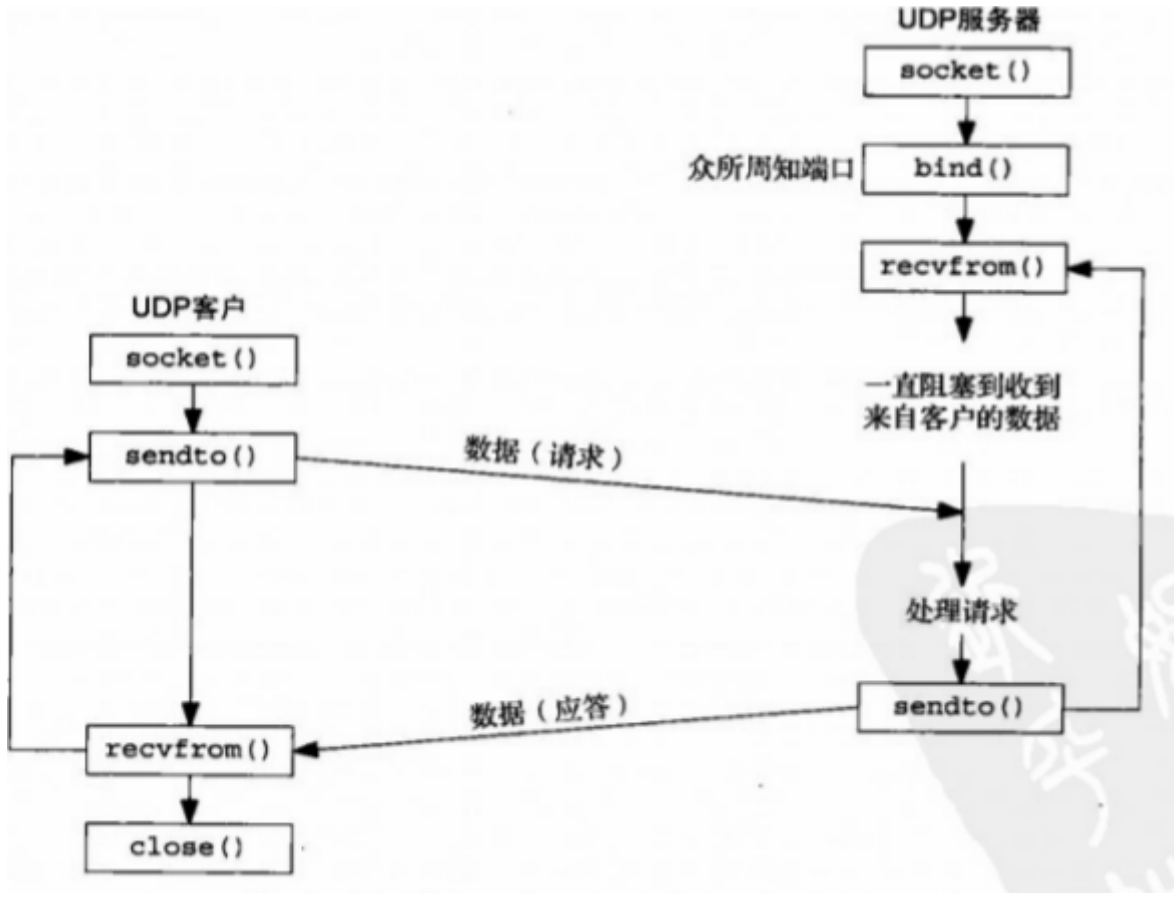
12. TCP 有 2 个套接字选项：

(1) TCP_MAXSEG：获取或设置 TCP 连接的最大分节大小（即 MSS）。

(2) TCP_NODELAY：禁止 TCP 的 Nagle 算法。默认情况下是启动的。

第 **8** 章 基本 **UDP** 套接字编程

1. 使用 UDP 编写的一些常见的应用程序有：DNS (域名系统)、NFS (网络文件系统）和 SNMP (简单网络管理协议）。（UDP、DNS、NFS、SNMP）
2. 在 UDP 客户/服务器程序中，客户不与服务器建立连接，而是只管使用 sendto 函数给服务器发送数据报，其中必须指定目的地（即服务器）的地址作为参数。类似地，服务器不接受来自客户的连接，而是只管调用 recvfrom 函数，等待来自某个客户的数据到达。recvfrom 将与所接收的数据报一道返回客户的协议地址，因此服务器可以把响应发送给正确的客户。（sendto()、recvfrom()）
3. UDP 调用关系：



4. 服务器应答数据报丢失可能会造成客户永远阻塞，为了防止这一点，一般方法是给客户的 recvfrom 调用设置一个超时。（recvfrom()、超时）

5. 可以给 UDP 套接字调用 connect ，然而这样做的结果却与 TCP 连接大相径庭：没有三路握手过程。（UDP、connect、没有三路握手）

- 未连接 UDP 套接字，新创建 UDP 套接字默认如此。
- 已连接 UDP 套接字，对 UDP 套接字调用 connect 的结果。

6. 对于已连接 UDP 套接字，与默认的未连接 UDP 套接字相比，发生了三个变化。

(1) 我们再也不能给输出操作指定的 IP 地址和端口号。也就是说，我们不使用 sendto，而改用 write 或 send。（已连接、不使用 sendto）

(2) 我们不必使用 recvfrom 以获悉数据报的发送者，而改用 read、recv 或 recvmsg。（不使用 recvfrom、用 read）

(3) 由已连接 UDP 套接字引发的异步错误会返回给它们所在的进程，而未连接 UDP 套接字不接收任何异步错误。（已连接、异步错误、返回所在进程）

第 **11** 章 名字与地址转换

1. 域名系统主要用于主机名字与 IP 地址之间的映射。主机名既可以是一个简单名字，例如 Solaris 或 bsdi，也可以是一个域名，例如 Solaris.unpbook.com。（主机名、域名）

2. DNS 主要用于主机名字与 IP 地址之间的映射。
3. DNS 中的条目称为资源记录：

- A：A 记录把一个主机名映射成一个 32 位的 IPv4 地址。（A 记录、IPv4）
- AAAA：AAAA 记录把一个主机名映射成 128 位的 IPv6 地址。（AAAA、IPv6）
- PTR：称为指针记录，把 IP 地址映射成主机名。（PTR、IP 地址、映射、主机名）
- MX：把一个主机指定作为给定主机的“邮件交换器”。　　（MX、邮件交换器）
- CNAME：代表“canonical name”（规范名字）。常见用法是为常用的服务（例如 ftp 和 www）指派 CNAME 记录。（CNAME、规范名字）

```
ftp     IN     CNAME  linux.unpbook.com.
www     IN     CNAME  linux.unpbook.com.
```

4. 在本书中编写的客户和服务器等应用程序通过调用称为解析器的函数库中的函数接触 DNS 服务器。常见的解析器函数是将在本章讲解的 gethostbyname 和 gethostbyaddr，前者把主机名映射成 IPv4 地址，后者则执行相反的映射。（gethostbyname、主机名映射成 IP、gethostbyaddr、相反映射）

   在这里，name 是主机名，addr 是 IP 地址。
5. gethostbyname()通过主机名来获取主机 IP。该函数只能返回 IPv4。该函数发生错误时，它不设置 errno 变量，而是将全局整数变量设置为相关值。（gethostbyname、错误、不设置 errno）
6. gethostbyaddr()通过 IP 地址来获取主机名，与 gethostbyname()的行为刚好相反。（gethostbyaddr()、获取主机名）
7. getservbyname()通过给定服务名获取对应的包含名字和服务号信息的 servent 结构指针。getservbyport()通过给定的端口号和协议名获取相关的服务信息。（getservbyname、获取 servent 结构）
8. getaddrinfo 函数把主机名字和服务名字映射到一个地址。获取指定地址的信息，返回值是一个结构 addrinfo 的链表。
9. 如果 getaddrinfo 失败，不能使用 perror()或 strerror()来生成错误消息，替代地，调用 gai_strerror()将返回的错误码转换成错误消息。（getaddinfo 失败、gai_strerror 生成错误消息）
10. 由 getaddrinfo ()返回的所有存储空间都是动态获取的，包括 addrinfo 结构体等，这些存储空间通过 freeaddrinfo()释放。（getaddrinfo、动态获取、freeaddrinfo）

第 **12** 章 **IPv4** 和 **IPv6** 的互操作性

1. 双栈，指 IPv4 协议栈和 IPv6 协议栈。（双栈、IPv4、IPv6）
2. 如果 IPv6 服务器收到来自某个 IPv4 客户的一个数据报，由 recvfrom 返回的地址将是该客户的 IPv4 映射的 IPv6 地址。服务器以这个 IPv4 映射的 IPv6 地址调用 sendto 给

出对本客户请求的响应。（IPv4 映射的 IPv6 地址）

3. IPv6 地址无法表示成一个 IPv4 地址，但 IPv4 地址可以映射成 IPv6 地址。
4. 客户和服务器的各种组合：

	IPv4服务器IPv4单栈主机（纯A）	IPv6服务器IPv6单栈主机（纯AAAA）	IPv4服务器双栈主机（A和AAAA）	IPv6服务器双栈主机（A和AAAA）
IPv4客户，IPv4单栈主机	IPv4	（无）	IPv4	IPv4
IPv6客户，IPv6单栈主机	（无）	IPv6	（无）	IPv6
IPv4客户，双栈主机	IPv4	（无）	IPv4	IPv4
IPv6客户，双栈主机	IPv4	IPv6	（无*）	IPv6

第 13 章 守护进程和 inetd 超级服务器

1. 守护进程也可能从某个终端由用户在 shell 提示符下键入命令行启动，这样的守护进程必须亲自脱离与控制终端的关联，从而避免与作业控制、终端会话管理、终端产生信号等发生任何不期望的交互，也可以避免在后台运行的守护进程非预期地输出到终端。
2. 守护进程有多种启动方法：（守护进程、多种启动方法）

(1) 在系统启动阶段，许多守护进程由系统初始化脚本启动。（初始化脚本启动）

(2) 许多网络服务器由将在本章靠后介绍的 inetd 超级服务器启动。　　（inetd 启动）

(3) cron 守护进程按照规则定期执行一些程序，而由它启动执行的程序同样作为守护进程运行。（cron 启动）

(4) at 命令用于指定将来某个时刻的程序执行。（at 启动）

(5) 守护进程还可以从用户终端或在前台或在后台启动。

3. 因为守护进程没有控制终端，所以当有事发生时它们得有输出消息的某种方法可用，而这些消息既可能是普通的通告性消息，也可能是需由系统管理员处理的紧急事件消息。syslog 函数是输出这些消息的标准方法，它把这些消息发送给 syslogd 守护进程。（没有控制终端、syslog、输出消息）
4. syslog()用于记录至系统记录。（syslog、记录至系统）
5. 通过调用 daemon_init 函数，能够把普通进程转变了守护进程：

```
#include	"unp.h"
#include	<syslog.h>

#define	MAXFD	64

extern int	daemon_proc; /* defined in error.c */

int
daemon_init(const char *pname, int facility)
{
```

```
    int             i;
    pid_t   pid;

    if ( (pid = Fork()) < 0)            // 复制进程
            return (-1);
    else if (pid)           // 退出父进程
            _exit(0);

    /* 子进程继续运行  */

    if (setsid() < 0)                       /* 创建新会话，那么子进程就变成会话的首进程  */
            return (-1);

    Signal(SIGHUP, SIG_IGN);
    if ( (pid = Fork()) < 0)    // 再次调用 fork，这时的父进程是上次 fork 的子进程
            return (-1);
    else if (pid)           // 退出父进程
            _exit(0);

    /* child 2 continues... */

    daemon_proc = 1;                        // 这个变量用于给其他函数使用

    chdir("/");                             // 更改工作目录

    /* close off file descriptors */
    for (i = 0; i < MAXFD; i++)
            close(i);           // 关闭继承来的所有的文件描述符

    /* redirect stdin, stdout, and stderr to /dev/null */
    open("/dev/null", O_RDONLY);    // 重定向标准输入、标准输出和标准出错
    open("/dev/null", O_RDWR);
    open("/dev/null", O_RDWR);

    openlog(pname, LOG_PID, facility);      // 打开 syslogd

    return (0);                             /* success */
}
```

但是，BSD 和 linux 提供了一个 daemon 函数用于实现类似的功能。

6. 4.3BSD 版本通过提供一个因特网超级服务器，即 inetd 守护进程。基于 TCP 或 UDP 的服务器都可以使用这个守护进程：（inetd 守护进程）

(1) 通过由 inetd 处理普通守护进程的大部分启动细节以简化守护程序的编写。这么一来每个服务器不再打调用 daemon_init 函数的必要。（简化守护程序的编写）

(2) 单个进程（inetd)就能为多个服务等待外来的客户请求，以此取代每个服务一个进程的做法。这么做减少了系统中的进程总数。（单个进程、等待、多个服务）

7. 启动一个程序并让它作为守护进程运行需要以下步骤：调用 fork()以转到后台运行，调用 setsid 建立一个新的 POSIX 会话并成为会话头进程，再次 fork()以避免无意中获得新的控制终端，改变工作目录和文件创建模式掩码，最后关闭所有非必要的描述符。

8. inetd 进程的配置文件是/etc/inetd.conf，它指定本超级服务器处理哪些服务以及当一个服务请求到达时该怎么做：（配置文件、/etc/inetd）

service-name	socket-type	protocol	wait-flag	login-name	server-program	server-program-arguments
ftp	stream	tcp	nowait	root	/usr/bin/ftpd	ftpd -1
telnet	stream	tcp	nowait	root	/usr/bin/telnetd	telnetd
login	stream	tcp	nowait	root	/usr/bin/rlogind	rlogind -s
tftp	dgram	udp	wait	nobody	/usr/bin/tftpd	tftpd -s /tftpboot

- service-name：服务名。必须在/etc/services 文件中定义。
- socket-type：对于 TCP ，该值为 stream；对于 UDP，该值为 dgram 。
- protocol：必须在/etc/protocols 文件中定义：TCP 或 UDP。
- wait-flag：对于 TCP 一般为 nowait，对于 UDP 一般为 wait。
- login-name：来自/etc/passwd 的用户名，一般为 root。
- server-program：调用 exec 指定的完整路径名。
- server-program -arguments：调用 exec 指定的命令行参数。

第 14 章 高级 I/O 函数

1. 在涉及套接字的 I/O 操作上设置超时的方法有以下 3 种：（超时、alarm、select、SO_RCVTIMEO 和 SO_SNDTIMEO 套接字选项）

(1) 调用 alarm，它在指定超时期满时产生 SIGALRM 信号。这个方法涉及信号处理，而信号处理在不同的实现上存在差异，而且可能干扰进程中现有的 alarm 调用。

(2) 在 select 中阻塞等待 I/O，以此代替直接阻塞在 read 或 write 调用上。也就是 select 的最后一个参数是愿意等待的时间。

(3) 使用较新的 SO_RCVTIMEO 和 SO_SNDTIMEO 套接字选项。这个方法的问题在于并非所有实现都支持这两个套接字选项。

```
struct timeval  tv;

tv.tv_sec = 5;
tv.tv_usec = 0;
Setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

2. 使用 alarm 函数和 SIGALRM 信号可以实现设置 I/O 操作的超时：（alarm、signal()）

```
sigfunc = Signal(SIGALRM, connect_alarm);
        if (alarm(nsec) != 0)
```

```
            err_msg("connect_timeo: alarm was already set");
```

signal()安装 SIGALRM 信号，然后使用 alarm()来设置定时器，定时器时间到，系统就会发送 SIGALRM 信号。（时间到、发送 SIGALRM 信号）

```
int
connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
        Sigfunc         *sigfunc;
        int             n;

        sigfunc = Signal(SIGALRM, connect_alarm);               // 安装 SIGALRM 信号
        if (alarm(nsec) != 0)                                   // 设置定时器时间
                err_msg("connect_timeo: alarm was already set");

        if ( (n = connect(sockfd, saptr, salen)) < 0) {         // 连接失败
                close(sockfd);                                  // 关闭文件描述符
                if (errno == EINTR)                             // 如果连接的过程中遇到中断，系统会将
errno 设置为 EINTR
                        errno = ETIMEDOUT;
        }
        alarm(0);                                               /* 关闭定时器  */
        Signal(SIGALRM, sigfunc); /* 恢复原来的信号处理函数  */

        return(n);
}
```

3. signal 函数返回信号处理程序之前的值，当发生错误时返回 SIG_ERR。所以可以用返回值来恢复原来的值：（signal()返回、之前的值）

```
        Sigfunc         *sigfunc;
        sigfunc = Signal(SIGALRM, connect_alarm);               // 安装 SIGALRM 信号
        ……
        Signal(SIGALRM, sigfunc); /* 恢复原来的信号处理函数  */
```

4. read、write 等系统调用在遇到中断时，系统会将 errno 设置为 EINTR。（中断、errno、设置为 EINTR）
5. 辅助数据即控制消息，可以通过调用 sendmsg 和 recvmsg 这两个函数，使用 msghdr 结构中的 msg_control 和 msg_controllen 这两个成员发送和接收。（辅助数据、控制消息、sendmsg、recvmsg）
6. 标准 I/O 函数库可用于套接字，不过需要考虑以下几点。

(1) 通过调用 fdopen，可以从任何一个描述符创建出一个标准 I/O 流。类似地，通过调用 fileno，可以获取一个给定标准 I/O 流对应的描述符。

(2) TCP 和 UDP 套接字是全双工的。标准 I/0 流也可以是全双工的：只要以 r+类型打开流即可，r+意味着读写。

(3) 解决上述读写问题的最简单方法是为一个给定套接字打开两个标准 I/O 流：一个用于读，一个用于写。

7. Solaris 上名为/dev/poll 的特殊文件提供了一个可扩展的轮询大量描述符的方法。
8. kqueue 接口允许进程向内核注册描述所关注 kqueue 事件的事件过滤器。事件除了与 select 所关注类似的文件 I/O 和超时外，还有异步 1/0、文件修改通知（例如文件被删除或修改时发出的通知）、进程跟踪（例如进程调用 exit 或 fork 时发出的通知）和信号处理。
9. kqueue 接口包括如下 2 个函数和 1 个宏：

- kqueue()：函数返回一个新的 kqueue 描述符，该描述符用于后面的 kevent()。
- kevent()：函数用于注册所关注的事件，或确定关注的事件是否发生。
- EV_SET()：是一个宏，用于用于初始化 kevent 结构，也就是 kenevt 的第二个参数。

10. T/TCP 是对 TCP 进行过略微修改的一个版木，能够避免近來彼此通信过的主机之间的三路握手。（T/TCP、修改版本、避免三路握手）
11. 在提供 T/TCP 的系统上 TCP 应用程序无需任何改动，除非要使用 T/TCP 的特性。所有现存 TCP 应用程序继续使用我们已经讲述过的套接字 API 工作。（T/TCP、无需任何改动）

第 15 章 Unix 域协议

1. Unix 域协议并不是一个实际的协议族，而是在单个主机上执行客户/服务器通信的一种方法，所用 API 就是在不同主机上执行客户/服务器通信所用的 API (套接字 API)。
2. Unix 域提供两类套接字：字节流套接字（类似 TCP）和数据报套接字（类似 UDP)。
3. Unix 域套接字地址结构为 sockaddr_un。

```
Struct sockaddr_un{
    sa_family_t sun_family;
    char        sun_path[104];
}
```

该结构体第二个成员是一个绝对路径地址，类似需要流式套接字要将套接字绑定到一个 IP 地址上，域套接字需要将套接字绑定到一个本地地址上。

4. socketpair()函数用于创建一对无名的、相互连接的套接字。本函数仅适用于 Unix 域套接字。
5. 当用于 Unix 域套接字时，套接字函数中存在一些差异和限制：

(1) 由 bind 创建的路径名默认访问权限应为 0777 (属主用户、组用户和其他用户都可读、可写并可执行），并按照当前 umask 值进行修正。（bind、0777、umask 修正）

(2) 与 Unix 域套接字关联的路径名应该是一个绝对路径名，而不是一个相对路径名。（关联路径、绝对路径名）

(3) 在 connect 调用中指定的路径名必须是一个当前绑定在某个打开的 Unix 域套接字上的路径名，而且它们的套接字类型（字节流或数据报）也必须一致。（路径名一致、套

接字类型一致）

(4) 调用 connect 连接一个 Unix 域套接字涉及的权限测试等同于调用 open 以只写方式访问相应的路径名。（权限测试、等同于、open 只写方式访问）

(5) Unix 域字节流套接字类似 TCP 套接字：它们都为进程提供一个无记录边界的字节流接口。

(6) 如果对于某个 Unix 域字节流套接字的 connect 调用发现这个监听套接字的队列已满，调用就立即返回一个 ECONNREFUSED 错误。

(7) Unix 域数据报套接字类似于 UDP 套接字：它们都提供一个保留记录边界的不可靠的数据报服务。

(8) 在一个未绑定的 Unix 域套接字上发送数据报不会自动给这个套接字捆绑一个路径名，这一点不同于 UDP 套接字：在一个未绑定的 UDP 套接字上发送 UDP 数据报导致给这个套接字捆绑一个临时端口。（未绑定、不会自动、UDP、会绑定）

6. 在两个进程之间传递描述符是通过把描述符作为辅助数据发送的。

第 16 章 非阻塞式 I/O

1. 套接字的默认状态是阻塞的。这就意味着当发出一个不能立即完成的套接字调用时，其进程将被投入睡眠，等待相应操作完成。可能阻塞的套接字调用可分为以下四类。（套接字、默认、阻塞）

(1) 输入操作，包括 read、readv、recv、recvfrom 和 recvmsg 共 5 个函数。

(2) 输出操作，包括 write、writev、send、sendto 和 sendmsg 共 5 个函数。

(3) 接受外来连接，即 accept 函数。

(4) 发起外出连接，即用于 TCP 的 connect 函数。

2. 对一个非阻塞的 TCP 套接字调用 connect，如果连接不能立即建立，那么连接的建立能照样发起（譬如送出 TCP 三路握手的第一个分组），不过会返回一个 EINPROGRESS 错误。

第 17 章 ioctl 操作

1. ioctl 函数传统上一直作为那些不适合归入其他精细定义的类别的特性的系统接口。ioctl 函数是 io 操作的杂物箱：（ioctl、io 操作的杂物箱）

```
int ioctl(int fd, int request, ... )
```

其中第三个参数总是一个指针，但指针的类型依赖于 request 参数。

2. 可以把 ioctl()相关的 request 划分为 6 类：（套接字操作、文件操作、接口操作、ARP 调整缓存、路由表、流系统）

- 套接字操作；
- 文件操作；
- 接口操作；

- ARP 高速缓存操作；
- 路由表操作；
- 流系统。

3. 明确用于套接字的 ioctl 请求有 3 个，它们都要求 ioctl 的第三个参数是指向某个整数的一个指针：

- SIOCATMARK：是否位于带外标记。
- SIOCGPGRP：设置套接字的进程 ID 或进程组 ID。
- SIOCSPGRP：获取套接字的进程 ID 或进程组 ID。

4. 用于文件的请求：

- FIONBIO：设置/清除非阻塞式 I/O 标志
- FIOASYNC ：设置/清除信号驱动异步 I/O 标志
- PIQNREAD：获取接收缓冲区的字节数
- FIOSETOWN：设置文件的进程 ID 或进程组 ID
- FIOGETOWN：获取文件地进程 ID 或进程组 ID

5. 操纵 ARP 高速缓存的 ioctl 请求有以下 3 个：

- SIOCSARP：把一个新的表项加到 ARP 高速缓存，或者修改其中已经存在的一个表项。
- SIOCDARP：从 ARP 高速缓存中删除一个表项。调用者指定要删除表项的网际网地址。
- SIOCGARP：从 ARP 高速缓存中获取一个表项。

  只有超级用户才能增加或删除表项。

6. 有些系统提供 2 个用于操纵路由表的 ioctl 请求。这 2 个请求要求 ioctl 的第三个参数是指向某个 rtentry 结构的一个指针：

- SIOCADDRT：路由表中增加一个表项。
- SIOCDELRT：从路由表中删除一个表项。

  没有办法查询路由表的所有表项。

7. 接口操作（通用接口）：

- SIOCGIFADDR：在 ifr_addr 成员中返回单播地址。
- SIOCSIFADDR：用 ifr_addr 成员设置接口地址。这个接口的初始化函数也被调用。
- SIOCGIFFLAGS：在 ifr_flags 成员中返回接口标志。
- SIOCSIFFLAGS：用 ifr_f lags 成员设置接口标志。
- SIOCGIFDSTADDR：在 ifr_dstaddr 成员中返回点到点地址。
- SIOCSIFDSTADDK：用 ifr_dstaddr 成员设置点到点地址。
- SIOCGIFBRDADDR：在 ifr_broadaddr 成员中返回广播地址。

- SIOCSIFBRDADDR：用 ifr_broadaddr 成员设置广播地址。
- SIOCGIFNETMASK：在 ifr_addr 成员中返回子网掩码。
- SIOCSIFNETMASK：用 ifr_addr 成员设置子网掩码。
- SIOCGIFMETRIC：用 ifr_metric 成员返回接口测度。
- SIOCSIFMETRIC：用 ifr_metric 成员设置接口的路由测度。

第 18 章 路由套接字

1. 内核中的 Unix 路由表传统上一直使用 ioctl 命令访问。（Unix 路由表、ioctl 命令访问）
2. 创建路由套接字：（AF_ROUTE、路由套接字）

```
socket(AF_ROUTE,SOCK_RAW,0);
```

创建路由套接字后，与其他套接字一样，可以调用 read() 或 write() 函数进行读写，但是由于系统内核是根据应用程序写入的消息来完成对路由信息的提供和修改的，因此，与其他套接字不同，写入和读出的数据都是有固定的格式的，例如：为了增加路由消息类型为 RTM_ADD；为了获取路由，消息类型为 RTM_GET。

3. 使用 sysctl()可以使不具有超级用户权限的程序能够获取路由表和接口列表信息。（sysctl、不具有权限、获取路由表信息）
4. 结构体 sockaddr_dl 是数据链路地址结构。（sockaddr_dl、数据链路地址结构）

第 20 章 广播

1. TCP 只支持单播寻址。
2. IPv6 往寻址体系结构中增加了任播方式。任播允许从一组通常提供相同服务的主机中选择一个（一般是选择按某种测度而言离源主机最近的）。通过适当地配置路由，并在多个位置往路由协议中注入同一个地址，多个 IPv4 或 IPv6 主机可以提供该地址的任播服务。
3. 多播支持在 IPv4 中是可选的，在 IPv6 中却是必需的。
4. IPv6 不支持广播。使用广播的任何 IPv4 应用程序一旦移植到 IPv6 就必须改用多播重新编写。
5. 广播和多播要求用于 UDP 或原始 IP，不能用于 TCP。
6. 广播的用途之一是在本地子网定位一个服务器主机，前提是已知或认定这个服务器主机位于本地子网，但不知道它的单播 IP 地址。这种方式也叫资源发现。如 DHCP、ARP。
7. 使用广播的例子：ARP、DHCP、网络时间协议 NTP、路由守护进程。
8. 我们可以使用记法{子网 ID,主机 ID}表示一个 IPv4 地址，其中子网 ID 表示由子网掩码覆盖的连续位，主机 ID 表示以外的位。如此表示的广播地址有以下两种，其中-1 表示所有位均为 1 的字段。

(1) 子网定向广播地址：{子网 ID,-1}。作为指定子网上所有接口的广播地址。举例来说，如果我们有一个 192.168.42/24 子网，那么 192.168.42.255 就是该子网上所有接口的子网定向广播地址。路由器通常不转发这种广播。

(2) 受限广播地址：{-1,-1}或 255.255.255.255。路由器从不转发目的地址为 255.255.255.255 的 IP 数据报。

注意：{子网 ID,主机 ID}指的是“子网 ID.主机 ID“例如 192.168.42.255 中的 192.168.42 就是子网 ID。

9. 应用进程无需就为接收广播 UDP 数据报而进行任何特殊处理：它只需要创建一个 UDP 套接字，并把应用的端口号捆绑到指定的端口号中。（接收 UDP 数据报、创建 UDP 套接字、捆绑到端口号）

10. 如果主机没有任何应用进程绑定指定的 UDP 端口。该主机的 UDP 代码于是丢弃这个接收取的数据报。该主机绝不能发送一个 ICMP 端口不可达消息，因为这么做可能产生广播风暴，即子网大量主机几乎同时产生一个响应，导致网络在一段时间内不可用。（指定端口没有进程、丢弃数据报、不发送不可达消息）

第 **21** 章 多播

1. 单播地址标识单个 IP 接口，广播地址标识某个子网的所有 IP 接口。单播和广播是寻址方案的两个极端（要么单个要么全部），多播则意在两者之间提供一种折衷方案。多播数据报只应该由对它感兴趣的接口接收，也就是说由运行相应多播会话应用系统的主机上的接口接收。另外，广播一般局限于局域网内使用，而多播则既可用于局域网，也可跨广域网使用。（多播、可在广域网使用）

2. 必须区分 IPv4 多播地址和 IPv6 多播地址：

(1) IPv4 的 D 类地址（从 224.0.0.0 到 239.255.255.255）是 IPv4 多播地址。D 类地址的低序 28 位构成多播组 ID (group ID)，整个 32 位地址则称为组地址。以下是特殊的 IPv4 多播地址：（IPv4、D 类地址、多播）

- 224.0.0.1 是所有主机组。子网上所有具有多播能力的节点（主机、路由器或打印机等）必须在所有具有多播能力的接口上加入该组。（224.0.0.1、所有主机组）
- 224.0.0.2 是所有路由器组。子网上所有多播路由器必须在所有具有多播能力的接口上加入该组。（224.0.0.2、所有路由器组）

(2) 下面是特殊的 IPv6 多播地址：

- ff0l::1 和 ff02::1 是所有节点组。类似于 IPv4 的 224.0.0.1 多播地址。但多播是 IPv6 的一个组成部分，这与 IPv4 是不同的。尽管对应的 IPv4 组称为所有主机组，而 IPv6 组称为所有节点组，它们的含义是一致的。
- ff01::2、ff02::2 和 ff05::2 是所有路由器组。子网上所有多播路由器必须在所有具有多播能力的接口上加入该组，类似于 IPv4 的 224.0.0.2 多播地址。

3. 在流式多媒体应用中，一个多播地址（IPv4 或 IPv6 地址）和一个传输层端口 (通常是 UDP 端口)的组合称为一个会话（session)。举例来说，一个音频/视频电话会议可能由两个会话构成：一个用于音频，一个用于视频。这些会话几乎总是使用不同的端

口，有时还使用不同的多播组，以便接收时灵活地选取。（地址、端口、会话、一个音频会议两个会话）

4. 广域网上的多播要使用多播路由器。（广域网、多播、多播路由器）
5. 广域网上的多播难以部署，最大的问题是多播地址的分配：IPv4 没有足够数量的多播地址可以静态地分配给想用的任何多播应用系统使用。源特定多播 SSM 则在有限程序上解决了这些问题。（广域网、多播难部署、多播地址的分配）
6. 源特定组播（SSM）是一种区别于传统组播的新的业务模型，它使用组播组地址和组播源地址同时来标识一个组播会话，而不是向传统的组播服务那样只使用组播组地址来标识一个组播会话。（源特定组播、SSM）
7. 传统意义的多播 API 支持只需要 5 个套接字选项。SSM 所需的源过滤额外要求多播 API 支持新增 4 个套接字选项。

第 22 章 UDP 套接字编程

1. 如果应用程序使用广播或多播，那就必须使用 UDP。（广播或多播、UDP）
2. 如果单个 TCP 连接用于多个请求-应答交换，那么连接的建立和拆除开销就由所有的请求和应答分担，这样的设计通常比为每个请求-应答交换使用新连接要好。尽管如此，有些应用系统还是为每个请求-应答交换使用一个新的 TCP 连接（例如较早版本的 HTTP)，而有些应用系统则在客户和服务器交换一个请求-应答后，可能数小时或数天不再通信，例如 DNS。
3. TCP 特性：（正面确认、丢失重传、流控制、慢启动）

- 正面确认，丢失分组重传，重复分组检测，给被网络打乱次序的分组排序。tcp 确认所有数据，以便检测出丢失的分组。
- 窗口式流控制。接收端 TCP 告知发送端自己已为接收数据分配了多大的缓冲区空间，发送端不能发送超过这个大小的数据。也就是说，发送端的未确认数据量不能超过接收端告知的窗口。
- 慢启动和拥塞避免。这是由发送端实施的一种流量控制形式，它通过检测当前的网络容量来应对阵发的拥塞。

4. 建议：

- 对于广播或多播应用程序必须使用 UDP。（广播或多播、UDP）
- 对于简单的请求-应答应用程序可以使用 UDP，不过错误检测功能必须加到应用程序内部。错误检测至少涉及确认、超时和重传。（简单请求应答、UDP、错误检测）
- 对于文件传输等海量数据传输不应该使用 UDP。（文件传输、不使用 UDP）

5. 如果想要让请求-应答应用程序使用 UDP，那么必须在客户程序中增加以下两个特性：（使用 UDP、增加两个特性）

- 超时和重传。（超时、重传）
- 序列号：供客户验证一个应答是否匹配相应的请求。（序列号）

6. 增加序列号比较简单。客户为每个请求冠以一个序列号，服务器必须在返送给客户的应答中回射这个序列号。这样客户就可以验证某个给定的应答是否匹配早先发出的请求。（回射序列号、匹配请求）

7. 处理超时和重传的老式方法是先发送一个请求并等待 N 秒钟。如果期间没有收到应答，那就重新发送同一个请求并再等待钟。如此发生一定次数后放弃发送。这是线性重传定时器的一个例了。

这个方法的问题在于数据报在网络上的往返时间可以从局域网的远不到一秒钟变化到广域网的好几秒钟。影响往返时间（RTT)的因素包括距离、网络速度和拥塞。

8. IPv6 为路径 MTU 控制提供了四个选项：

(1) 执行路径 MTU 发现时，IP 数据报通常按照外出接口 MTU 或路径 MTU 二者中较小者进行分片。使用最小 MTU 是由 IPV6_USE_MIN_MTU 套接字选项控制。

(2) 接收路径 MTU 变动：由 IPV6_RECVPATHMTU 套接字选项以接收路径 MTU 变动通知。

(3) IPV6_PATHMTU 套接字确定某个已连接套接字的当前路径 MTU。该选项只能获取。

(4) IPV6_DONTFRAG 选项用于关闭自动分片特性。

9. 应用进程可以开启 IPV6_RECVPATHMTU 套接字选项以接收路径 MTU 变动通知。本标志值使得任何时候路径 MTU 发生变动时作为辅助数据由 recvmsg 返回变动后的路径 MTU。（IPV6_RECVPATHMTU、接收路径变动通知）

10. IPV6_DONTFRAG 套接字选项用于关闭自动分片特性；默认值为 0，表示允许自动分片，为 1 则关闭自动分片。（IPV6_DONTFRAG、关闭自动分片）

第 **24** 章 带外数据

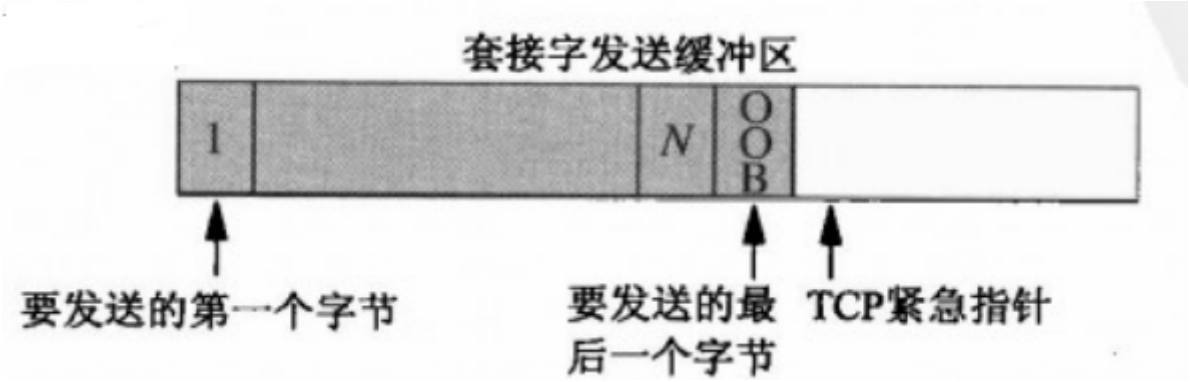
1. 带外数据 OOB 有时也称为经加速数据。其想法是一个连接的某端发生了重要的事情，而且该端希望迅速通告其对端。这里”迅速“意味着这种通知应该在已经排队等待发送的任何“普通”（有时称为“带内”）数据之前发送。也就是说，带外数据被认为具有比普通数据更高的优先级。带外数据并不要求在客户和服务器之间再使用一个连接，而是被映射到已有的连接中。（带外数据、重要事件、通知对端）

2. UDP 没有实现带外数据。 （UDP、没有带外数据）

3. 以 MSG_OOB 标志调用 send 函数写出一个含有 ASCII 字符 a 的单字节带外数据：（MSG_OOB 标志）

```
send(fd,"a",1,MSG_OOB);
```

TCP 把这个数据放置在该套接字发送缓冲区的下一个可用位置，并把该连接的 TCP 紧急指针设置成再下一个可用位置。



4. TCP 首部指出发送端已经进入紧急模式（即伴随紧急偏移的 URG 标志已经设置），但是由紧急指针所指的实际数据字节却不一定随同送出。即使发送端 TCP 因流量控制而暂停发送数据（接收端的套接字接收缓冲区已满，导致其 TCP 向发送端 TCP 通告了一个值为 0 的窗口），紧急通知照样不伴随任何数据地发送出去。（紧急模式、不一定随同送出）
5. 当有 OOB 数据到来时，服务端进程会引发 SIGURG 信号，在服务器进程里可以捕捉这个信号并处理。

```
Signal(SIGURG, sig_urg);
Fcntl(connfd, F_SETOWN, getpid());
```

Fcntl()一句用于设置套接字 connfd 所有者为当前进程。

6. TCP 支持紧急标记的概念：在普通数据流中紧急数据所在的位置。为帮助数据是否接收到紧急标记，可以使用 sockafmark()。当下一个要读取的字节在紧急标志处时，sockatmark()返回 1。

- 带外标记总是指向普通数据最后一个字节之后的位置。这意味着，如果带外数据在线接收，那么如果下一个待读入的字节是使用 MSG_OOB 标志发送的，sockatmark 就返回真。而如果 SO_OOBINLINE 套接字选项没有开启，那么，若下一个待读入的字节是跟在带外数据后发送的第一个字节，sockatmark 就返回真。
- 读操作总是停在带外标记上。也就是说，如果在套接字接收缓冲区中有 100 个字节，不过在带外标记之前只有 5 个字节，而进程执行一个请求 100 个字节的 read 调用，那么系统强制返回带外标记之前的 5 个字节，而不是读 100 个字节。这种在带外标记上强制停止读操作的做法使得进程能够调用 sockatmark 确定缓冲区指针是否处于带外标记。

示例：发送端是 tcpsend04.c，先发普通数据“123”，接着带外数据“4”，最后发普通数据“5”：

```
int
main(int argc, char **argv)
{
        int                 listenfd, connfd, n, on=1;
        char    buff[100];

        if (argc == 2)
```

```
        listenfd = Tcp_listen(NULL, argv[1], NULL);          // 正常的
sock()、bind()、listen()流程
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], NULL);
    else
        err_quit("usage: tcprecv04 [ <host> ] <port#>");

    Setsockopt(listenfd, SOL_SOCKET, SO_OOBINLINE, &on, sizeof(on));      // 开启
SO_OOBINLINE 选项，在这情况下不能使用 MSG_OOB 的方式来读带外数据

    connfd = Accept(listenfd, NULL, NULL);
    sleep(5);

    for ( ; ; ) {
        if (Sockatmark(connfd))                    // 是否接收到带外数据
            printf("at OOB mark\n");

        if ( (n = Read(connfd, buff, sizeof(buff)-1)) == 0) {  // 第一次读到的是 123，之后
收到带外数据 4，读出来，接着读到 5
            printf("received EOF\n");
            exit(0);
        }
        buff[n] = 0;    /* null terminate */
        printf("read %d bytes: %s\n", n, buff);
    }
}
```

输出如下：

```
read 3 bytes: 123
at OOB mark
read 2 bytes：  45
recvived EOF
```

7. 即使因为流量控制而停止发送数据了，TCP 仍然发送带外数据的通知（即它的紧急指针）。
8. 在带外数据到达之前，接收进程可能被通知说发送进程已经使用 SIGURG 信号或通过 select 发送了带外数据。如果接收进程接送指定 MSG_OOB 调用 recv，而带外数据却尚未到达，recv 将返回 EWOULDBLOCK 错误。
9. 如果在接收进程读入某个现有带外数据之前有新的带外数据到达，先前的标记就丢失。
10. 心搏函数，也就是心跳函数，是判断 TCP 连接是否已经断开的函数。示例实现原理：客户端定时发送一个 MSG_OOB，服务器收到 MSG_OOB 后回射信号，如果客户端能收到服务器回射 MSG_OOB 信号，则说明连接没有断开。服务器端也定时发送信号验证是否断开。
11. TCP 本身有保持存活特性，也就是 SO_KEEPALIVE 套接字选项，但该选项在等待时间过长，连接空置两个小时后才发送一个存活探测段。要将时间改小，则不能使用该

选项。（SO_KEEPALIVE 选项、时间过长）

第 25 章 信号驱动式 I/O

1. 信号驱动式 I/O，即 SIGIO，是指进程预先告知内核，使得当某个描述符上发生某事时，内核使用信号通知相关进程。（信号驱动式 I/O、预告告知内核）
2. 针对一个套接字使用信号驱动式 I/O 要求进程执行以下 3 个步骤。（信号驱动式 I/O、3 个步骤）

(1) 必须注册一个响应 SIGIO 的信号回调函数

(2) 通过 fcntl 设置 F_SETOWN，使得 socket 属于某个进程

(3) 通过 fcntl 设置 O_ASYNC 将该 socket 设置为异步

```
Signal(SIGURG, sig_urg);
 Fcntl(connfd, F_SETOWN, getpid());

void sig_urg(int signo)
{
   ……
}
```

3. 在 UDP 上使用信号驱动式 I/O 是简单的。SIGIO 信号在发生以下事件时产生：

(1) 数据报到达套接字；

(2) 套接字上发生异步错误。

因此当捕获对于某个 UDP 套接字的 SIGIO 信号时，我们调用 recvfrom 或者读入到达的数据报，或者获取发生的异步错误。

4. 信号驱动式 I/O 对 TCP 套接字近乎无用，该信号产生太过频繁。（信号驱动式 I/O、TCP、无用）

第 26 章 线程

1. fork 的代价太高，而线程可以解决这个问题。
2. 同一进程内的所有线程除了共享全局变量外还共享：（共享指令、数据、打开的文件、信号处理函数、工作目录、用户 ID）

- 进程指令。
- 大多数数据。
- 打开的文件（即描述符)。
- 信号处理函数和信号处置。
- 当前工作目录。
- 用户 ID 和组 ID。

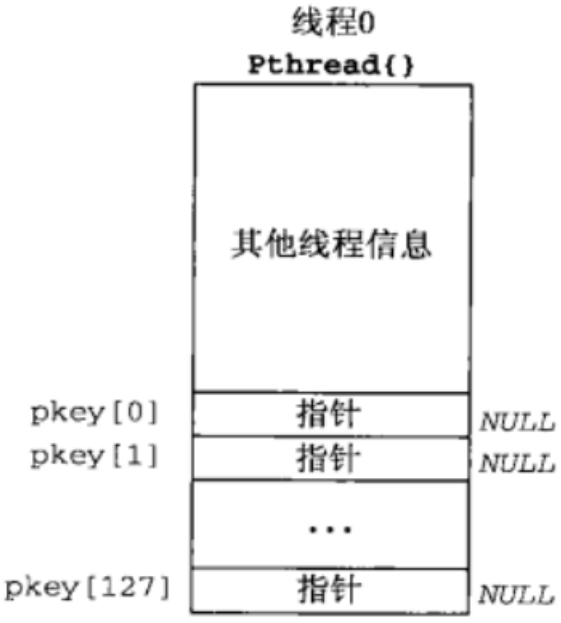
3. 每个线程有各自的：（各自的、线程 ID、寄存器、栈、errno、信号掩码、优先级）

- 线程 ID。
- 寄存器集合，包括程序计数器和栈指针。
- 栈（用于存放局部变量和返回地址)。
- errno。
- 信号掩码。
- 优先级。

4. 线程使用静态变量时，不同线程无法为自己保存各自的变量值。（静态变量、无法保存、各自的值）
5. 系统为每个进程维护一个称为 key 结构的结构数组：

Key 结构中的标志指示这个数组元素是否正在使用，所有的标志初始化为“不在使用”。当一个线程调用 pthread_key_create 创建一个新的线程特定数据元素时，系统搜索其 Key 结构数组找出第一个不在使用的元素。该元素的索引（0～127）称为健，返回给调用线程的正是这个索引。（key 标志、pthread_key_create、找出第一个不在使用）

6. 除了进程范围的 Key 结构数组外，系统还在进程内维护关于每个线程的多条信息。这些特定于线程的信息我们称之为 Pthread 结构，其部分内容是我们称之为 pkey 数组的一个 128 个元素的指针数组。



pkey 数组的所有元素都被初始化为空指针。这些 128 个指针是和进程内的 128 个可能的“键”逐一关联的值。（pkey 指针、键关联的值）

7. 当我们调用 pthread_key_create 创建一个键时，系统告诉我们这个键（索引）。每个线程可以随后为该键存储一个值（指针），而这个指针通常又是每个线程通过调用 malloc 获得的。线程特定数据中易于混淆的地方之一是：该指针是键-值对中的值，但是真正的线程特定数据却是该指针指向的任何内容。（创建键、存储一个值）

第 27 章 IP 选项

1. IPv4 允许在 20 字节首部固定部分之后跟以最多共 40 个字节的选项。尽管已经定义的 IPv4 选项共有 10 种，最常用的却是源路径选项。（IPv4、最常用、源路径选项）
2. 读取和设置 IP 选项使用 getsockopt 和 setsockopt 函数。（读取和设置、IP 选项、getsockopt、setsockopt）
3. 源路径是由 IP 数据报的发送者指定的一个 IP 地址列表。如果源路径是严格的（strict)，那么数据报必须且只能逐一经过所列的节点。也就是说列在源路径中的所有节点必须前后互为邻居。如果源路径是宽松的（loose），那么数据报必须逐一经过所列的节点，不过也可以经过未列在源路径中的其他节点。（源路径、经过的节点 IP 列表）

第 28 章 原始套接字

1. 通常情况下程序员接所接触到的套接字（Socket）为两类：

(1) 流式套接字（SOCK_STREAM）：一种面向连接的 Socket，针对于面向连接的 TCP 服务应用；

(2) 数据报式套接字（SOCK_DGRAM）：一种无连接的 Socket，对应于无连接的 UDP 服务应用。

2. 但是，当我们面对如下问题时，SOCK_STREAM、SOCK_DGRAM 将显得这样无助：

(1) 怎样发送一个自定义的 IP 包？

(2) 怎样发送一个 ICMP 协议包？

(3) 怎样分析所有经过网络的包，而不管这样包是否是发给自己的？

(4) 怎样伪装本地的 IP 地址？

原始套接字（SOCK_RAW）可以用来自行组装数据包，可以接收本机网卡上所有的数据帧（数据包），对于监听网络流量和分析网络数据很有作用。原始套接字广泛应用于高级网络编程，也是一种广泛的黑客手段。著名的网络 sniffer（一种基于被动侦听原理的网络分析方式）、拒绝服务攻击（DOS）、IP 欺骗等都可以通过原始套接字实现。

3. 原始套接字可以收发内核没有处理的数据包。（原始、内核没有处理的数据包）
4. 原始套接字提供普通的 TCP 和 UDP 套接字所不提供的以下 3 个能力：

- 有了原始套接字，进程可以读与写 ICMPv4、IGMPv4 和 ICMPv6 等分组。举例来说，ping 程序就使用原始套接字发送 ICMP 回射请求并接收 ICMP 回射应答。（读写 ICMPv4 等分组）
- 有了原始套接字，进程可以读写内核不处理其协议字段的 IPv4 数据报。（不处理协议字段）
- 有了原始套接字，进程还可以使用 IP_HDRINCL 套接字选项自行构造 IPv4 首部。这个能力可用于构造譬如说 TCP 或 UDP 分组。（自定义 IPv4 首部）

5. 只有超级用户才能创建原始套接字，这么做可防止普通用户往网络写出它们自行构造器的 IP 数据报。
6. 创建一个原始套接字涉及如下步骤。

(1) 把第二个参数指定为 SOCK_RAW 并调用 socket 函数，以创建一个原始套接字。第三个参数（协议）通常不为 0。举例来说，我们使用如下代码创建一个 IPv4 原始套接字：（SOCK_RAW）

```
int sockfd;
sockfd = socket (AF_INET, SOCK_RAW, protocol);
```

其中 protocol 参数是形如 IPPROTO_xxx 的某个常值，定义在<netinet/in. h>头文件中，如 IPPROTO_IGMP。

(2) 可以在这个原始套接字上按以下方式开启 IP_HDRINCL 套接字选项：（开启 IP_HDRINCL）

```
const int on = 1;
if(setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
    (出错处理)
```

(3) 可以在这个原始套接字上调用 bind 函数，不过比较少见。bind 函数仅仅设置本地地址，因为原始套接字不存在端口号的概念。

(4) 可以在这个原始套接字上调用 connect 函数，不过也比较少见。connect 函数仅仅设置外地地址，同样因为原始套接字不存在端口号的概念。

7. 内核把哪些接收到的 IP 数据报传递到原始套接字？这儿遵循如下规则。

- 接收到的 UDP 分组和 TCP 分组绝不传递到任何原始套接字。如果一个进程想要读取含有 UDP 分组或 TCP 分组的 IP 数据报，它就必须在数据链路层读取这些分组。（UDP、TCP 分组、不传递）
- 大多数 ICMP 分组在内核处理完其中的 ICMP 消息后传递到原始套接字。（ICMP、传递）
- 所有 IGMP 分组在内核完成处理其中的 IGMP 消息后传递到原始套接字。（IGMP、传递）
- 内核不认识其协议字段的所有 IP 数据报传递到原始套接字。内核对这些分组执行的

唯一处理是针对某些 IP 首部字段的最小验证：IP 版本、IPv4 首部校验和、首部长度以及目的 1P 地址。（不认识、传递）

- 如果某个数据报以片段形式到达，那么在它的所有片段均到达且重组出该数据报之前，不传递任何片段分组到原始套接字。（以片段形式、传递）

8. 当内核有一个需传递到原始套接字的 IP 数据报时，它将检查所有进程上的所有原始套接字，以寻找所有匹配的套接字。每个匹配的套接字将被递送以该 IP 数据报的一个副本。内核对每个原始套接字均执行如下 3 个测试，只有这 3 个测试结果均为真，内核才把接收到的数据报递送到这个套接字。（套接字、3 个测试、为真、才递送）

- 如果创建这个原始套接字时指定了非 0 的协议参数（socket 的第三个参数），那么接收到的数据报的协议字段必须匹配该值，否则该数据报不递送到这个套接字。（非 0 参数、匹配该值）
- 如果这个原始套接字已由 bind 调用绑定了某个本地 IP 地址，那么接收到的数据报的目的 IP 地址必须匹配这个绑定地址，否则该数据报不递送到这个套接字。（匹配本地 IP）
- 如果这个原始套接字已由 connect 调用指定了某个外地 IP 地址，那么接收到的数据报的源 IP 地址必须匹配这个已连接地址，杏则该数据报不递送到这个套接字。（匹配外地 IP）

第 29 章 数据链路访问

1. 原始套接口使得我们可以读写内核不处理的 IP 数据报，而对数据链路层访问则把这种能力进一步扩大——读写任何类型的数据链路帧，而不仅仅是 IP 数据报。
2. Unix 上访问数据链路层的 3 个常用方法是 BSD 的分组过滤器 BPF、SVR4 的数据链路提供者接口 DLPI 和 Linux 的 SOCK_PACKET 接口。（数据链路层、BPF、数据链路提供者接口 DLPI、SOCK_PACKET 接口）
3. BPF 的强大威力却在于它的过滤能力。打开一个 BPF 设备的每个应用进程可以装载各自的过滤器，这个过滤器随后由 BPF 应用于每个分组。有些过滤器比较简单（例如“udp or tcp”只接收 UDP 或 TCP 分组），不过更复杂的过滤器可以检查分组首部某些字段是否为特定值。（BPF、过滤能力、UDP、只接收 UDP）
4. libnet 函数库提供构造任意协议的分组并将其输出到网络中的接口。它以与实现无关的方式提供原始套接字访问方式和数据链路访问方式。（libnet、构造分组）
5. Linux 先后有两个从数据链路层接收分组的方法。较旧的方法是创建类型为 SOCK_PACKET 的套接字，这个方法的可用面较宽，不过缺乏灵活性。较新的方法创建协议族为 PF_PACKET 的套接字，这个方法引入了更多的过滤和性能特性。举例来说，从数据链路接收所有帧应如下创建套接字：（数据链路、接收分组、SOCK_PACKET、PF_PACKET）

```
fd = socket (PF_PACKET, SOCK_RAW, htons {ETH_P_ALL));    /*  较新方法  */

或
```

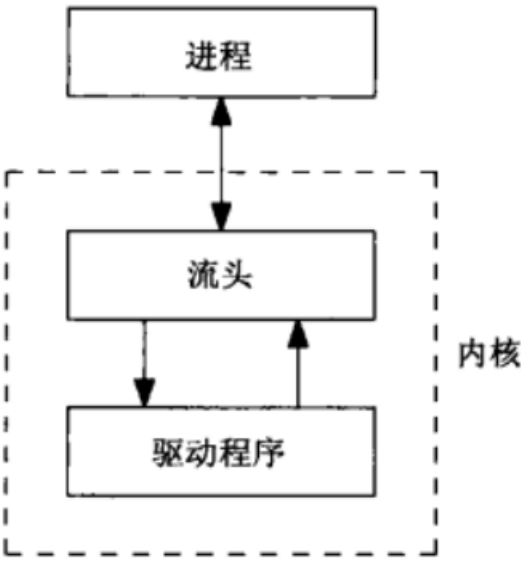
```
fd = socket (AF_INET,SOCK_PACKET, htons (ETH_P_ALL) ) ;    /*  较旧方法  */
```

如果只想捕获 IPv4 帧，那就如下创建套接字：

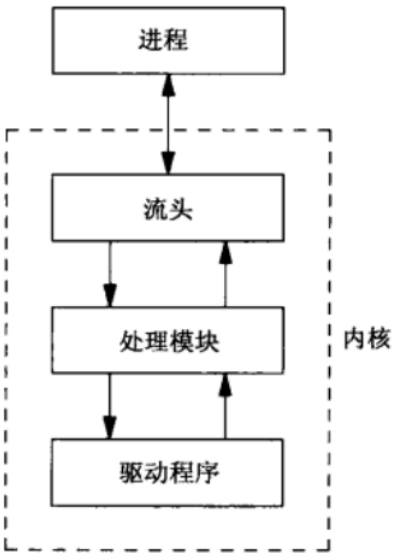
```
fd = socket (PF_PACKET, SOCK_RAW, htons(ETH_P_IP)) ;      /* 较新方法  */

或

fd = socket (AF_INET,SOCK_PACKET, htons (KTH_P_IP)) ;     /* 较旧方法  */
```

第 31 章 流

1. 流在进程和驱动程序之间提供全双工的连接，如图所示。虽然我们称底部那个方框为驱动程序，它却不必与某个硬件设备相关联，也就是说它可以是一个伪设备驱动程序(即软件驱动程序)。　　（流、全双工）



2. 可以在流头下方插入任意数量的模块：（流头下方、插入）



3. 多路复选器是一种特殊类型的伪设备驱动程序，它从多个源接受数据。（多路复选器、多个源接收数据）
4. 流消息可划分为高优先级、优先级带和普通三类。优先级共有 256 带，在 0～255 之间取值，其中普通消息位于带 0。流消息的优先级用于排队和流量控制。按约定高优先级消息不受流量控制影响。（高优先级、优先级带、普通、256 带）

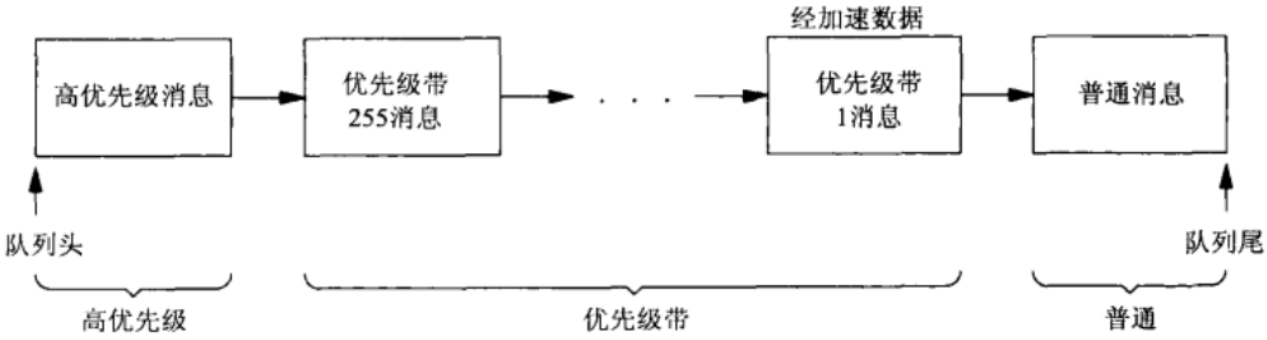


图31-5　一个队列中的流消息基于优先级的排序

5. getmsg()用于从流中获取控制信息或数据，也可以同时发送两者。putmsg()用于向流中发送控制信息或数据，也可以同时发送两者。（getmsg、获取控制信息或数据、putmsg、发送控制信息或数据）

Unix 网络编程：卷二

第一部分 简介

第 1 章 简介

1. 我们可以把任意类型的 IPC 的持续性定义成该类型的一个对象一直存在多长时间：（进程持续、内核持续、文件系统持续）

- 随进程持续的。IPC 对象一直存在到打开着该对象的最后一个进程关闭该对象为止。例如管道和 FIFO 就是这种对象。（进程持续、管道、FIFO）
- 随内核持续的。IPC 对象一直存在到内核重新自举或显式删除该对象为止。例如 SystemV 的消息队列、信号量和共享内存区就是此类对象。Posix 的消息队列、信号

量和共享内存区必须至少是随内核持续的，但也可以是随文件系统持续的，具体取决于实现。（内核持续、Posix IPC）

- 随文件系统持续的。IPC 对象一直存在到显式删除该对象为止。即使内核重新自举了，该对象还是保持其值。Posix 消息队列、信号量和共享内存区如果是使用映射文件实现的（不是必需条件），那么它们就是随文件系统持续的。（文件系统持续）

IPC类型	持续性
管道	随进程
FIFO	随进程
Posix互斥锁	随进程
Posix条件变量	随进程
Posix读写锁	随进程
fcntl记录上锁	随进程
Posix消息队列	随内核
Posix有名信号量	随内核
Posix基于内存的信号量	随进程
Posix共享内存区	随内核
System V消息队列	随内核
System V信号量	随内核
System V共享内存区	随内核
TCP套接字	随进程
UDP套接字	随进程
Unix域套接字	随进程

2. 当两个或多个无亲缘关系的进程使用某种类型的 IPC 对象来彼此交换信息时，该 IPC 对象必须有一个某种形式的名字（name)或标识符（identifier)，这样其中一个进程（往往是服务器）可以创建该 IPC 对象，其余进程则可以指定同一个 IPC 对象。（无亲缘、IPC、名字或标识符）

IPC类型	用于打开或创建IPC的名字空间	IPC打开后的标识
管道	（没有名字）	描述符
FIFO	路径名	描述符
Posix互斥锁	（没有名字）	pthread_mutex_t指针
Posix条件变量	（没有名字）	pthread_cond_t指针
Posix读写锁	（没有名字）	pthread_rwlock_t指针
fcntl记录上锁	路径名	描述符
Posix消息队列	Posix IPC名字	mqd_t值
Posix有名信号量	Posix IPC名字	sem_t指针
Posix基于内存的信号量	（没有名字）	sem_t指针
Posix共享内存区	Posix IPC名字	描述符
System V消息队列	key_t键	System V IPC标识符
System V信号量	key_t键	System V IPC标识符
System V共享内存区	key_t键	System V IPC标识符
门	路径名	描述符
Sun RPC	程序/版本	RPC句柄
TCP套接字	IP地址与TCP端口	描述符
UDP套接字	IP地址与UDP端口	描述符
Unix域套接字	路径名	描述符

3. 管道没有名字，因此不能用于无亲缘关系的进程间，但是 FIFO 有一个在文件系统中的 Unix 路径名作为其标识符，因此可用于无亲缘关系的进程间。（管道、没有名字、FIFO、用于无亲缘）

4. 包裹函数执行实际函数函数，同时在实际函数的基础上判断该函数的返回值，是否出错。（包裹函数、判断返回值）

```
387 void
388 Sem_post(sem_t *sem)
389 {
390     if (sem_post(sem) == -1)
391         err_sys("sem_post error");
392 }
```

   这里约定包裹函数以大写字母开头。

5. 线程函数出错时并不设置标准的 Unix errno 变量；相反，本该设置 errno 的值改由线程函数作为其返回值返回调用者。这意味着我们每次调用任意一个线程函数时，都得分配一个变量来保存函数返回值，然后在调用我们的 err_sys 函数前，把 errno 设置成所保存的值。（线程函数、不设置 errno、返回 errno 值）

```
int     n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

6. 每当在个 Unix 函数中发生错误时，全局变量 errno 将被设置成一个指示错误类型的正数，函数本身则通常返回-1。errno 的值只在某个函数发生错误时设置。如果该函数不返回错误，errno 的值就无定义。（发生错误、errno）
7. Posix 是“可移植操作系统接口”（Portable Operating System Interface）的首字母缩写。并不是一个单一标准，而是一系列标准。（Posix、可移植、接口）

第 2 章 Posix IPC

1. 以下三种类型的 IPC 合称为“Posix IPC”：（Posix IPC、消息队列、信号量、共享内存）

- Posix 消息队列
- Posix 信号量
- Posix 共享内存区

2. Posix 相关函数：

	消息队列	信号量	共享内存区
头文件	<mqueue.h>	<semaphore.h>	<sys/mman.h>
创建、打开或删除IPC的函数	mq_open mq_close mq_unlink	sem_open sem_close sem_unlink	shm_open shm_unlink
		sem_init sem_destroy	
控制IPC操作的函数	mq_getattr mq_setattr		ftruncate fstat
IPC操作函数	mq_send mq_receive mq_notify	sem_wait sem_trywait sem_post sem_getvalue	mmap munmap

3. Posix IPC 都使用“PosixIPC 名字”进行标识。mq_open、sen_open 和 shm_open 这三个函数的第一个参数就是这样的一个名字，它可能是某个文件系统中的一个真正的路径名，也可能不是。（Posix IPC、名字）
4. Posix.1 是这么描述 Posix IPC 名字的：（符合路径规则、斜杠开头）

- 它必须符合已有的路径名规则（必须最多由 PATH_MAX 个字节构成，包括结尾的空字节）。
- 如果它以斜杠符开头，那么对这些函数的不同调用将访问同一个队列。如果它不以斜杠符开头，那么效果取决于实现。
- 名字中额外的斜杠符的解释由实现定义。

5. mq_open()、sem_open()和 shm_open()这三个分别是创建或打开一个 IPC 对象的函数。（mq_open、sem_open、shm_open、创建或打开）
6. mq_open、sem_open 或 shm_open 函数的 O_CREAT 标志用于创建新的消息队列、有名信号量或共享内存区对象。（O_CREAT、创建）

第 3 章 System V IPC

1. 以下三种类型的 IPC 合称为 System V IPC：（System V IPC、消息队列、信号量、共享内存）

- System V 消息队列
- System V 信号量
- System V 共享内存区

2. System V IPC 相关函数：

	消息队列	信号量	共享内存区
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
创建或打开IPC的函数	msgget	semget	shmget
控制IPC操作的函数	msgctl	semctl	shmctl
IPC操作函数	msgsnd msgrcv	semop	shmat shmdt

3. 三种类型的 System V IPC 使用 key_t 值作为它们的名字。key_t 为一个整数，至少 32 位。函数 ftok 把一个已存在的路径名和一个整数标识符转换成一个 key_t 值，称为 IPC 键。（System V IPC、key_t、名字、ftok、转换成键）

4. 内核为每个 IPC 对象维护一个信息结构，其内容跟内核给文件维护的信息类似：（IPC 对象、ipc_perm）

```
struct ipc_perm {
  uid_t    uid;    /* owner's user id */
  gid_t    gid;    /* owner's group id */
  uid_t    cuid;   /* creator's user id */
  gid_t    cgid;   /* creator's group id */
  mode_t   mode;   /* read-write permissions */
  ulong_t  seq;    /* slot usage sequence number */
  key_t    key;    /* IPC key */
};
```

5. ipc_perm 结构还含有一个名为 seq 的变量，它是一个槽位使用情况序列号。该变量是一个由内核为系统中每个潜在的 IPC 对象维护的计数器。每当删除一个 IPC 对象时，内核就递增相应的槽位号，若溢出则循环回 0。（seq、槽位使用情况）

也就是说，每当删除一个 IPC 对象再重用时，它的文件描述符值应为原文件描述符加上槽位号 seq。（删除再重用、描述符值加 seq）

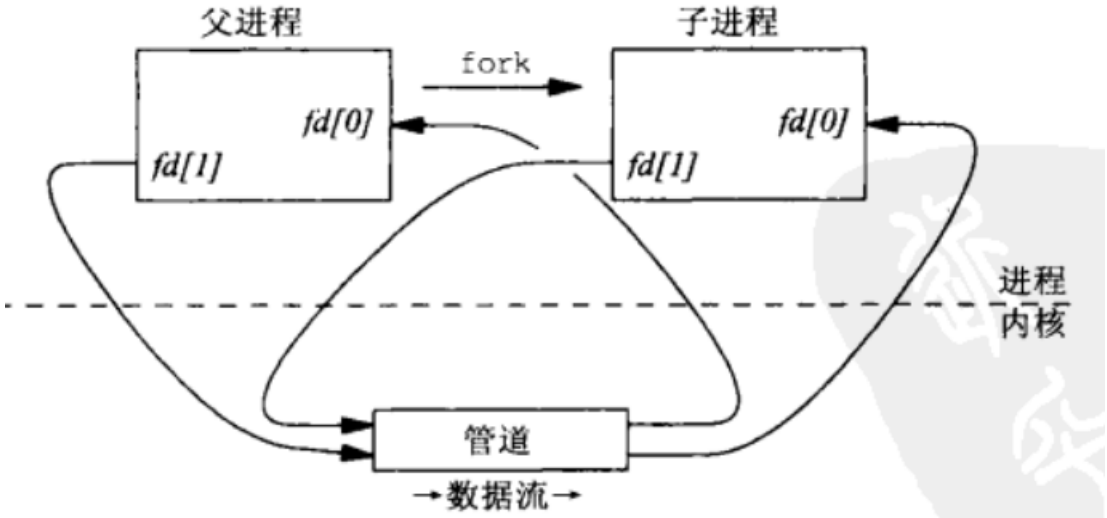
6. 实现这些 IPC 的任何系统都提供了两个特殊的程序：ipcs 和 ipcrm。ipcs 输出有关 system V IPC 特性的各种信息，ipcrm 则删除一个 System V 消息队列、信号量集或共享内存区。（ipcs、输出 IPC 特性信息、ipcrm、删除 IPC）

7. 文件描述符只有特定的进程中有用，不同的进程同一个数值的文件描述符可能指不同的文件。（文件描述符、特定进程中有用）

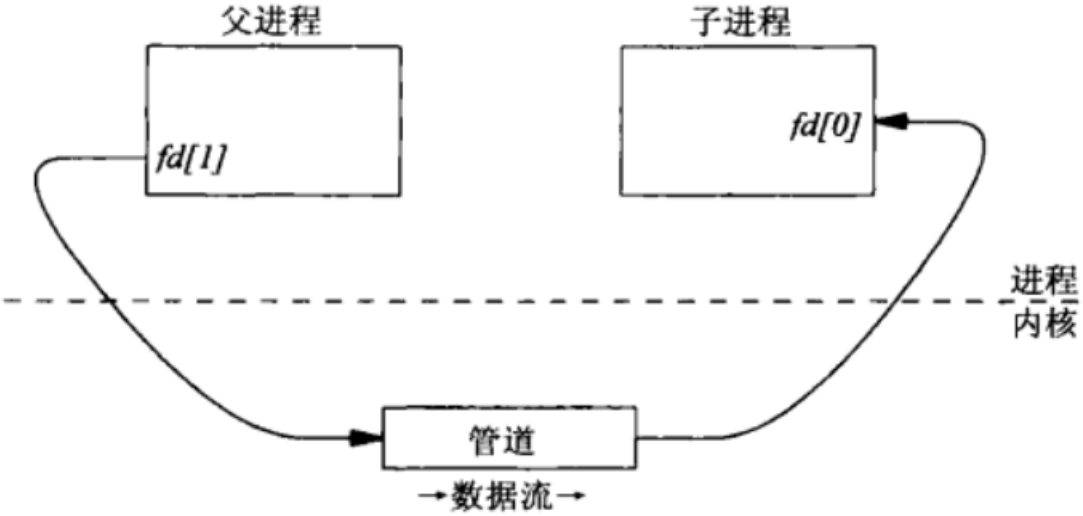
第二部分 消息传递

第 **4** 章 管道和 **FIFO**

1. 管道是最初的 Unix IPC 形式，缺点是没有名字，而 FIFO 是管道的修正版，有时称为有名管道。管道和 FIFO 都是使用通常的 read 和 write 函数访问的。（管道、没有名字、FIFO、修改版）
2. 管道使用 pipe()创建。（pipe()、创建管道）
3. 管道的典型用途是以下述方式为两个不同进程提供进程间的通信手段。



父进程关闭读端口，子进程关闭写端口。（父关读、子关写）



4. 当半双工管道需要一个双向数据流时，我们必须创建两个管道，每个方向一个。

5. 函数 popen()会调用 fork()产生子进程，然后从子进程中调用/bin/sh -c 来执行参数 command 的指令。这个进程必须由 pclose() 函数关闭，而不是 fclose() 函数。（popen()、产生子进程、执行 command、pclose()、关闭）
6. FIFO 由 mkfifo 函数创建。（FIFO、mkfifo 创建）
7. 一个描述符能以两种方式设置成非阻塞：（设置非阻塞、O_NONBLOCK 标志、调用 fcntl）

(1) 调用 open 时可指定 O_NONBLOCK 标志。

(2) 如果一个描述符已经打开，那么可以调用 fcntl 以启用 O_NONBLOCK 标志。对于管道来说，必须使用这种技术，因为管道没有 open 调用，在 pipe 调用中也无法指定 O_NONBLOCK 标志。

8. 当对一个管道或 FIFO 的最终 close 发生时，该管道或 FIFO 中任何残余数据都被丢弃。（close、残余数据被丢弃）
9. FIFO 只能在单台主机上使用，不能通过 NFS 安装到文件系统上。（FIFO、单台主机）
10. 恶意客户发送一个请求行，但不打开自己的 FIFO 来读，这属于拒绝服务型攻击。（发送请求、不打开 FIFO 读、攻击）
11. 系统加于管道和 FIFO 的唯一限制为：（限制、OPEN_MAX、最大描述符数、PIPE_BUF、最大数据量）

- OPEN_MAX：一个进程在任意时刻打开的最大描述符数；
- PIPE_BUF：可原子地写往一个管道或 FIFO 的最大数据量。

使用 sysconf()可以输出这些限制。（sysconf、输出限制）

第 5 章 Posix 消息队列

1. Posix 消息队列相关函数使用前缀 mq_XXX()，而 System V 消息队列相关函数使用前缀 msXXX()。
2. 消息队列可认为是一个消息链表。有足够写权限的线程可往队列中放置消息，有足够读权限的线程可从队列中取走消息。（消息队列、消息链表）
3. Posix 消息队列和 System V 消息队列的差别：（消息队列、差别）

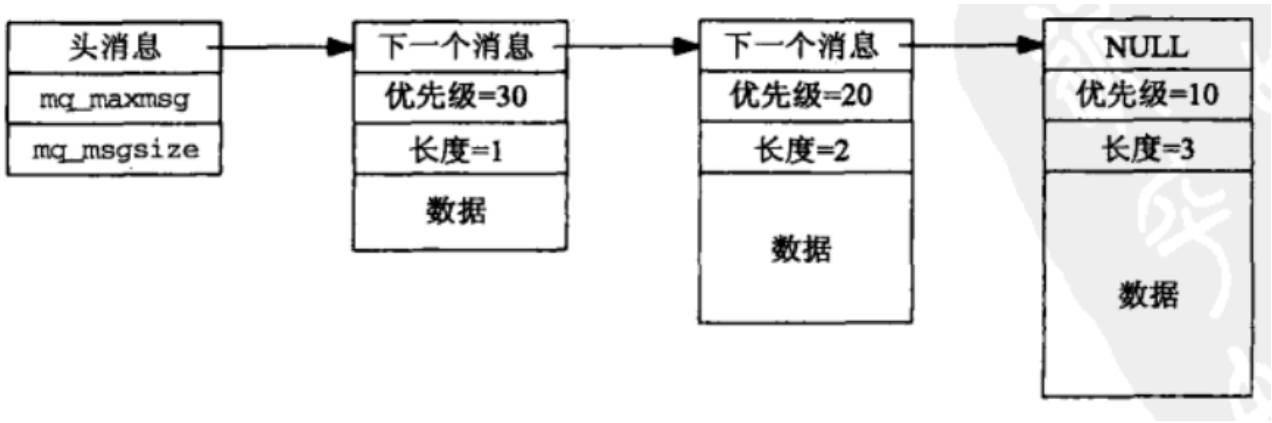
- 对 Posix 消息队列的读总是返回最高优先级的最早消息，对 System V 消息队列的读则可以返回任意指定优先级的消息。（Posix、读、返回最高、最早）
- 当往一个空队列放置一个消息时，Posix 消息队列允许产生一个信号或启动一个线程，System V 消息队列则不提供类似机制。（Posix、空队列放置、允许产生或启动）

4. 队列中的每个消息具有如下属性：

- 一个无符号整数优先级（Posix）或一个长整数类型（SystemV）；
- 消息的数据部分长度（可以为 0)；

- 数据本身（如果长度大于 0)。

5. 含有三个消息的某个 Posix 消息队列的可能布局：



6. mq_open()创建一个新的消息队列或打开一个已存在的消息队列。（mq_open()、创建或打开）
7. 打开的消息队列是由 mq_close()关闭的。（mq_close、关闭）
8. 要从系统中删除用作 mq_open 第一个参数的某个 name，必须调用 mq_unlink()。每个消息队列有一个保存其当前打开着描述符数的引用计数器，因而本函数能够实现类似于 unlink 函数删除一个文件的机制。（mq_unlink()、删除）
9. 每个消息队列有四个属性，mq_getattr 把所指定队列的当前属性填入由 attr 指向的结构，mq_setattr 则给所指定的队列设置属性，但只使用由 attr 指向的 mq_attr 结构的 mq_flags 成员，以设置或清除非阻塞标志。（mq_getattr()、获取属性、mq_setattr()、设置属性）

```
Struct mq_attr{
    long mq_flags;      # 消息队列标志，0 表示不阻塞
    long mq_maxmsg;  # 消息队列允许的最大长度
    long mq_msgsize;  # 消息的最大字节数
    long mq_curmsgs;   当前队列的消息数量
}
```

10. mq_send()往队列中放置一个消息。mq_receive()从一个队列中取走一个消息。（mq_send()、放置消息、mq_receive()、取走消息）
11. 任意给定队列的两个限制，它们都是在创建该队列时建立的：（队列限制、最大消息数、最大字节数）

- mq_mqxmsg：队列中的最大消息数
- mq_msgsize：给定消息的最大字节数

12. 消息队列的实现定义了另外两个限制：

- MQ_OPEN_MAX：一个进程能够同时拥有的打开着消息列表的最大数目。
- MQ_PRIO_MAX：任意消息的最大优先级值加 1。

13. Posix 消息队列允许异步事件通知，以告知何时有一个消息放置到某个空消息队列中。这种通知有两种方式可供选择：（消息队列、异步通知、告知有消息）

- 产生一个信号。
- 创建一个线程来执行指定的函数。

这种通知通过调用 mq_motify()建立，该函数为指定队列建立或删除异步事件通知。（mq_motify()、建立或删除、异步通知）

14. 异步信号安全指可以从信号处理程序中调用的函数。下表列出这些 Posix 函数：（异步信号安全、从信号处理程序中调用）

access	fpathconf	rename	sysconf
aio_return	fstat	rmdir	tcdrain
aio_suspend	fsync	sem_post	tcflow
alarm	getegid	setgid	tcflush
cfgetispeed	geteuid	setpgid	tcgetattr
cfgetospeed	getgid	setsid	tcgetpgrp
cfsetispeed	getgroups	setuid	tcsendbreak
cfsetospeed	getpgrp	sigaction	tcsetattr
chdir	getpid	sigaddset	tcsetpgrp
chmod	getppid	sigdelset	time
chown	getuid	sigemptyset	timer_getoverrun
clock_gettime	kill	sigfillset	timer_gettime
close	link	sigismember	timer_settime
creat	lseek	signal	times
dup	mkdir	sigpause	umask
dup2	mkfifo	sigpending	uname
execle	open	sigprocmask	unlink
execve	pathconf	sigqueue	utime
_exit	pause	sigset	wait
fcntl	pipe	sigsuspend	waitpid
fdatasync	raise	sleep	write
fork	read	stat	

没有列在该表中的函数不可以从信号处理程序中调用。

15. 信号可划分为两个大组：（信号、两大类）

(1) 其值在 SIGRTMIN 和 SIGRTMAX 之间（包括两者在内）的实时信号。Posix 要求至少提供 RTSIG_MAX 实时信号，而该常值的最小值为 8。（SIGRTMIN、SIGRTMAX、之间、实时信号）

(2) 所有其他信号：SIGALRM、SIGINT、SIGKILL 等。（其他信号）

16. 接收某个信号的进程的 sigaction 调用是否指定了新的 SA_SIGINFO 标志很关键。如果此标志指定了，那么实时信号的实时行为有保证，否则实时行为未指定，但对于一般信号来说，无论此标志有无指定，实时行为都未指定。（指定 SA_SIGINFO、实时行

为有保证）

17. 实时行为有如下特征：（实时行为）

(1) 信号是排队的，同一信号产生了三次，那么它就递交三次。（产生三次、递交三次）

(2) 当有多个 SIGRTMIN 到 SIGRTMAX 范围内的解阻塞信号排队时，值越小的优先权越高。（解阻塞、越小、优先权越高）

(3) 当某个非实时信号递交时，传递给它的信号处理程序的唯一参数就是该信号的值；但对于实时信号来说，它可以携带更多的信息。例如，通过 SA_SIGINFO 标志安装的实时信号的信号处理程序声明如下：

```
void func(int signo, siginfo_t *info, void *content)

typedef struct {
      int si_signo; /*same value as signo argument*/
      int si_code; /*SI_{USER,QUEUE,TIMER,ASYNCIO,MEGEQ}*/
      union sigval si_value; /*integer or pointer value from sender*/
} siginfo_t;
```

(4) 一些新的函数定义成使用实时信号工作。例如，sigqueue 函数代替 kill 函数。

18. siginfo_t 结构体中的 si_code 值，此值是用来表明此信号是由哪个事件产生的，列表如下：

- SI_ASYNCIO:信号是由某个异步 I/O 请求完成。
- SI_MESGQ:信号是由一个消息放到一个空的队列上时产生的。
- SI_QUEUE：信号是由 sigqueue 函数产生的。
- SI_TIMER：信号是由使用 timer_settime 函数设置的某个定时器到时产生的。
- SI_USER：信号是由 kill 函数产生的。

第 6 章 System V 消息队列

1. System V 消息队列使用消息队列标识符标识。具有足够特权的任何进程都可以往一个给定队列放置一个消息，具有足够特权的任何进程都可以从一个给定队列读出一个消息。（System V 消息队列、标识符标识）
2. msget()用于创建一个新的消息队列或访问一个已存在的消息队列。（msget()、创建或访问）
3. 使用 msgsnd()将数据放到消息队列中。（msgsnd()、放到消息队列）
4. 使用 msgrcv()从消息队列中取用消息。（msgrcv()、取出消息）
5. 使用 msgctl()对消息队列执行多种操作。（msgctl()、多种操作）
6. System V 消息队列的问题之一是它们由各自的标识符而不是描述符标识，这意味着我们不能在消息队列上直接使用 select 或 poll。（不能、直接使用 select 或 poll）

解决该问题的办法之一是：让服务器创建一个管道，然后派生一个子进程，由子进程阻塞在 msgrcv 调用中。当有一个消息准备好被处理时，msgrcv 返回，子进程接着从所指定的队列中读出该消息，并把该消息写入管道。服务器父进程当时可能在该管道以及一些网络连接上 select。这种办法的负面效果是消息被处理了三次：一次是在子进程使用 msgrcv 读出时，一次是在子进程进入管道时，最后一次是在父进程从该管道中读出时。为避免这样的额外处理，父进程可以创建一个在它自身和子进程之间分享的共享内存区，然后把管道用作父子进程间的一种标志。（子进程、子进程阻塞、msgrcv()）

7. 消息队列的限制：（限制、最大字节数、最大消息队列数、最大消息数）

- msgmax：每个消息的最大字节数
- msgmnb：任何一个消息队列上的最大字节数
- msgmni：系统范围的最大消息队列数
- msgtql：系统范围的最大消息数

第三部分 同步

第 7 章 互斥锁和条件变量

1. 互斥锁指代相互排斥，它是最基本的同步形式。互斥锁用于保护临界区，以保证任何时文只有一个线程在执行其中的代码，或者任何时间只有一个进程在执行其中的代码。（互斥锁、相互排斥）
2. 保护一个临界区的代码通常轮廓大体如下：（轮廓大体）

```
lock_the_mutex(…);
临界区
unlock_the_mutex(…)
```

3. Posix 互斥锁互斥锁被声明为具有 pthread_mutex_t 数据类型的变量。如果互斥锁变量是静态分配的，那么我们可以把它初始化成常值 PTHREAD_MUTEX_INITIALIZER。例如：　　（posix 互斥锁、pthread_mutex_t 类型、静态、常值 PTHREAD_MUTEX_INITIALIZER）

```
Static pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
```

如果互斥锁是动态分配的，或者分配在共享内存中，那么我们必须在运行之时通过调用 pthread_mutex_init 函数来初始化它。（动态分配、pthread_mutex_init()）

4. pthread_mutex_lock()用于对互斥量进行加锁。pthread_mutex_trylock()用于尝试对互斥量进行加锁。pthread_mutex_unlock()用于解锁互斥量。（pthread_mutex_lock()、加锁、pthread_mutex_trylock()、尝试加锁、pthread_mutex_unlock()、解锁）
5. 互斥锁用于上锁，条件变量用于等待。（互斥锁、上锁、条件变量、等待）
6. pthread_cond_wait()：等待条件变为真，如果在给定时间内条件不满足，则生成一个代码出错码的返回变量。（pthread_cond_wait()、等待条件变真）

   pthread_cond_timedwait()：和 pthread_cond_wait()相似，只是多了个 timeout 参数，指

定等待时间。（pthread_cond_timedwait()、相似 pthread_cond_wait()）

pthread_cond_signal()：唤醒等待该条件的线程。（pthread_cond_signal()、唤醒）

pthread_cond_broadcast()：唤醒等待该条件的所有线程。（pthread_cond_broadcast()：唤醒所有）

第 8 章 读写锁

1. 读写锁的数据类型为 pthread_rwlock_t。如果这个类型的某个变量是静态分配的，那么可通过给它赋常值 PTHREAD_RWLOCK_INITIALIZER 来初始化它。（读写锁、pthread_rwlock_t、静态分配、常值、PTHREAD_RWLOCK_INITIALIZER）

第 9 章 记录上锁

1. 记录锁指的是锁住文件的某一部分，而非整个文件。（记录锁、锁住文件一部分）
2. 粒度用于标记能被锁住的对象的大小。对于 Posix 记录上锁来说，粒度就是单个字节。通常情况下粒度越小，允许同时使用的用户数就越多。比如粒度为文件或记录。（粒度、文件粒度、记录粒度）
3. 使用 fcntl 函数来实现添加记录锁。（fcntl()、添加记录锁）
4. Posix 记录锁是建议性锁，它不能防止一个进程写已由另一个进程读锁定的文件。（记录锁、建议性锁）
5. 建议性锁只在 cooperating processes 之间才有用，对 cooperating process 的理解是最重要的，它指的是会影响其它进程的进程或被别的进程所影响的进程，举个例子：（我们可以同时在两个窗口中运行同一个命令，对同一个文件进行操作，那么这两个进程就是 cooperating processes。（建议性锁、cooperating processes 间才有用）
6. 强制性锁是另一种类型的记录锁。使用强制性锁后，内核检查每个 read 和 write 请求，以验证其操作不会干扰由某个进程持有的某个锁。（强制性锁、另一种记录锁）
7. 对某个特定文件施行强制性锁，应满足：

- 组成员执行位必须关掉。
- SGID 位必须打开。

8. 强制性上锁不需要新的系统调用。

第 10 章 Posix 信号量

1. Posix 信号量相关函数使用前缀 sem_XXX()，System V 信号量相关函数使用 semXXX()。
2. 信号量：（有名信号量、基于内存的信号量、System V 信号量）

- Posix 有名信号量：使用 Posix IPC 名字标识，可用于进程或线程间的同步。

- Posix 基于内存的信号量：存放在共享内存中，可用于进程或线程间的同步。
- System V 信号量：在内核中维护，可用于进程或线程间的同步。

3. 二值信号量：其值或为 0 或为 1 的信号量。（二值信号量、0 或 1）
4. 一个进程可以在某个信号量上执行三种操作：（三种操作、创建、等待、挂出）

- 创建。要求调用者指定初始值。
- 等待。该操作测试信号量的值，如果其值小于或等于 0，那就等待，一旦其值变为大于 0 就将它减 1。
- 挂出。该操作将信号量的值加 1。（挂出、加 1）

5. sem_open()用于创建一个新的有名信号量或打开一个已存在的有名信号量。有名信号量总是既可用于线程间的同步，又可用于进程间的同步。（sem_open()、创建或打开）
6. sem_close()关闭信号量。（sem_close()、关闭）
7. 关闭信号量并没有将它从系统中删除。Posix 有名信号量至少是随内核持续的：即使当前没有进程打开着某个信号量，它的值仍然保持。（关闭、没有删除）
8. 使用 sem_unlink()将有名信号量从系统中删除。（sem_unlink()、删除）
9. sem_wait()：如果信号量计数是 0，则阻塞，直到成功使信号量减 1。（sem_wait()、为 0、则阻塞）

   sem_trywait()：如果信号量是 0，不阻塞，而是返回 1-，并将 errno 置为 EAGAIN。

10. sem_post()：将 POSIX 信号量的值加 1。（sem_post()、值加 1）

    sem_getvalue()：获取信号量的值。成功后，valp 指向的整数值将包含信号量的值。（sem_getvalue()、获取值）

11. sem_init()：初始化一个未命名的 POSIX 信号量。（sem_init()、初始化）

    sem_destroy()：对未命名的信号量使用完成时，调用该函数销毁它。（sem_destroy、销毁）

12. 信号量定义了两个限制：（信号量、限制、最大信号量数、最大值）

- SEM_NSEMS_MAX：一个进程可同时打开着最大信号量数。
- SEM_VALUE_MAX：一个信号量的最大值。

第 11 章 System V 信号量

1. 对于系统中的每个信号量集，内核维护一个如下的信息结构：（信号量集、信息结构、semid_ds）

```
Struct semid_ds{
    struct ipc_perm sem_perm;
    struct sem      *sem_base;
    ushort          sem_nsems;
```

```
    time_t          sem_otime;
    time_t          sem_ctime;
}
```

sem 结构是内核用于维护某个给定信号量的一组值的内部数据结构。sem_base 含有指向某个 sem 结构数组的指针：当前信号量集中的每个信号量对应其中一个数组元素。

2. semget()：创建一个信号量或访问一个已存在的信号量集。（semget()、创建或访问）
3. semop()：对一个或多个信号量进行操作。（semop()、对信号量进行操作）
4. semctl()：对一个信号量执行各种控制操作。（semctl()、各种控制操作）
5. System V 信号量系统限制：

名　字	说　明	DUnix 4.0B	Solaris 2.6
semmni	系统范围最大信号量集数	16	10
semms1	每个信号量集最大信号量数	25	25
semmns	系统范围最大信号量数	400	60
semopm	每个semop调用最大操作数	10	10
semmnu	系统范围最大复旧结构数①		30
semume	每个复旧结构最大复旧项数	10	10
semvmx	任何信号量的最大值	32767	32767
semaem	最大退出时调整（adjust-on-exit）值	16384	16384

第四部分 共享内存区

第 12 章 共享内存区介绍

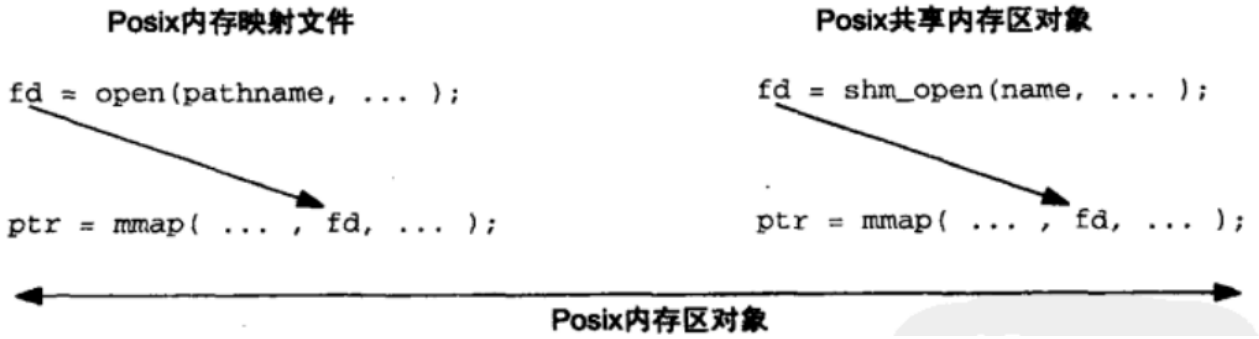
1. 共享内存区是可用 IPC 形式中最快的。一旦这样的内存区映射到共享它的进程的地址空间，这些进程间数据的传递就不再涉及内核。（共享内存、最快）
2. mmap()：将一个给定的文件映射到一个存储区域中。（mmap()、映射）
3. munmap()：解除映射区。（munmap()、解除映射）
4. msync()：将缓冲区数据写回存储设备，它并不等待实际写磁盘操作结束。该函数作用于存储映射区。（msync()、缓冲区、写回设置）
5. 并不是所有文件都能进程内存映射。例如，访问终端或套接字的描述符就无法 mmap。该类型的描述符必须使用 read 和 write 来访问。（有些文件不能进行内存映射）
6. mmap 的另一个用途是在无亲缘关系的进程间提供共享的内存区。（mmap()、无亲缘关系、共享内存）

第 13 章 Posix 共享内存区

1. Posix 共享内存相关函数使用前缀 shm_XXX()，而 System V 共享内存相关函数使用前缀 shmXXX()。
2. Posix 提供了两种在无亲缘关系进程间共享内存区的方法：（两种、无亲缘关系、共享内存）

(1) 内存映射文件：由 open 函数打开，由 mmap 函数把得到的描述符映射到当前进程地址空间中的一个文件。（内存映射文件、open()打开）

(2) 共享内存区对象：由 shm_open 打开一个 Posix.1IPC 名字，所返回的描述符由 mmap 函数映射到当前进程的地址空间。（共享内存区对象、shm_open()打开）



3. Posix 共享内存区涉及以下两个步骤要求：

(1) 指定一个名字参数调用 shm_open，以创建一个新的共享内存区对象或打开一个已存在的共享内存区对象。

(2) 调用 mmap 把这个共享内存区映射到调用进程的地址空间。

4. shm_open()：创建一个新的共享内存区对象或打开一个已存在的共享内存区对象。（shm_open()、创建或打开）

   shm_unlink()：删除一个共享内存区对象。（shm_unlink()、删除）

5. ftruncate()：将参数 fd 指定的文件大小改为参数 length 指定的大小。 如果原来的文件大小比参数 length 大，则超过的部分会被删除。（ftruncata()、修改大小）
6. fstat()：获取已在描述符 fildes 上打开文件的有关信息。（fstat()、获取信息）

第 14 章 System V 共享内存区

1. System V 共享内存区在概念上类似于 Posix 共享内存区。代之以调用 shm_open()后调用 mmap 的是，先调用 shmget()，再调用 shmat()。（System V、先 shmget()、再 shmat()）
2. 内核为每个共享存储段维护着一个 shmid_ds 结构。（shmid_ds 结构）
3. shmget()：创建一个新的共享内存区，或者访问一个已存在的共享内存区。（shmget()、创建或访问）
4. shmat()：一旦创建了共享存储段，进程可以用 shmat()将其连接到它的地址空间中。

（shmat()、连接到地址空间）

5. shmdt()：当对共享存储段操作已经结束时，调用 shmdt()脱离该段。（shmdt()、脱离共享存储段）
6. shmctl()：对共享存储执行多种操作。（shmctl()、多种操作）
7. 共享内存区限制：

名　字	说　明	DUnix 4.0B	Solaris 2.6
shmmax	一个共享内存区的最大字节数	4 194 304	1 048 576
shmmnb	一个共享内存区的最小字节数	1	1
shmmni	系统范围最大共享内存区标识符数	128	100
shmseg	每个进程附接的最大共享内存区数	32	6

图14-5　System V共享内存区的典型系统限制

第五部分 远程过程调用

第 15 章 门

1. 本地过程调用：C 编程中经常使用的过程调用，被调用的过程与调用过程处于同一个进程。（本地过程调用、同一个进程）

   远程过程调用：被调用过程和调用过程处于不同的进程。（远程过程调用、不同进程）

2. RPC 通常允许一台主机的某个客户调用另一台主机上的某个服务器过程，只要这两台主机以某种形式的网络连接着。
3. 门：一个进程调用同一台主机上另一个进程中的某个过程（函数）。（门、调用另一个进程、过程）
4. 在进程内部，门是用描述符标识的。在进程以外，门可能是用文件系统中的路径名标识的。（门、描述符标识）
5. door_call()：由客户调用，它会调用在服务器进程的地址空间中执行的一个服务器过程。（door_call()、客户调用、调用服务器过程）
6. door_create()：由服务器调用，调用该函数创建一个服务器过程。（door_create()、服务器调用、创建）
7. 门调用是同步的：当客户调用 door_call 时，该函数直到服务器过程返回时才返回。（门调用、服务器过程返回、才返回）
8. door_return()：服务器过程完成工作时通过调用该函数返回。这会使用客户中相关联的 door_call 调用返回。（door_return()、服务器、完成时、调用该函数返回）
9. door_cred()：获取每个调用对应的客户信息。（door_cred()、获取客户信息）

10. door_info()：客户通过该函数找出有关服务器的信息。（door_info()、获取服务器信息）
11. 通过一个门从客户向服务器传递描述符的手段是，把 door_arg_t 结构的 desc_ptr 成员设置成指向一个 door_desc_t 结构的数组，door_num 成员则设置成这些 door_desc_t 结构的数目。从服务器向客户传递回描述符的手段是，把 door__return 的第三个参数设置成指向一个 door_desc_t 结构的数组，把该函数的第四个参数设置成待传递描述符的数目。
12. 当客户请求到达时，门函数库会按照处理它们的需要自动地创建新线程。这些线程由该函数库作为脱离的线程创建，具有默认的线程栈大小，禁止了线程取消功能，并具有从调用 door_create 的线程初始继承来的信号掩码和调度类。如果我们想要改变上述任何特性，或者希望亲自管理服务器线程池，那么可以调用 door_server_create()以指定我们自己的服务器创建过程。（请求到达、自动创建新线程、door_server_create()、指定、服务器创建过程）
13. door_bind()：将调用线程捆绑到与指定的门关联的私有服务器池中。如果调用线程已绑定在另外某个门上，那就执行一个隐匿的松绑操作。（door_bind()、线程、绑定到私有服务器池）

door_unbind()：显式地把调用线程从其已绑定的门上松绑。（door_unbind()、松绑）

door_revoke()：撤销对于由 fd 标识的门的访问。一个门描述符只能由创建它的进程撤销。（door_revoke()、撤销）

14. 等势一词用在描述一个过程时，意思是该过程可调用任意多次而不出问题。返回当前时间和日期的过程是等势的，但银行账户减去一笔费用的过程是非等势的。（等势、调用多次、不出问题）
15. 当客户和服务器分散到两个进程上时，必须考虑任何要方崩溃时会发生什么事情。（两个进程、崩溃时会发生）

- 服务器过早终止，那么客户的 door_call()将返回一个 EINTR 错误。（过早终止、EINTR 错误）
- door_call()不是一个可重新启动的系统调用，例如 door_call()被中断后，函数返回 EINTR。（不可重新启动、EINTR 错误）
- 客户第一次 door_call()被中断手，第二次 door_call 调用启动了再次调用服务器过程的另一个线程。如果该服务器过程是等势的，那是没有问题。但如果该服务器过程是非等势的，那就有问题了。（第二次调用、等势、没问题）
- 客户中止：当系统检测到客户在其 door_call()仍然进行期间即将终止时，就向处理该调用的服务器线程发送一个取消请求。（客户中止、发送取消请求）

如果该服务器线程禁止了取消功能，那就什么事都不发生，该线程继续执行到执行完成，结果则被丢弃。

如果该服务器线程启用了取消功能，那就调用所设置的任何清理处理程序，该线程随后终止。

第 16 章 Sun RPC

1. 显式网络编程：直接调用 socket()、connect()等套接字 API 或 XTI API。（显式网络编程、套接字 API）

   隐式网络编程：类似远程过程调用（RPC）这样的工具。（隐式网络编程、RPC）

2. 远程过程调用是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。

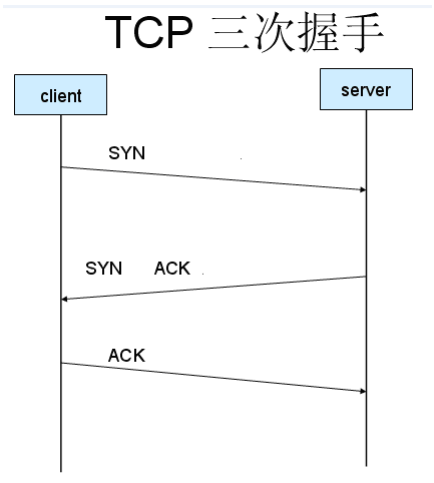
整理

1. wait()：进程一旦调用了 wait，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子 进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。

   参数 status 用来保存 被收集进程退出时的一些状态。

```
/*********************************************
* 这是一个信号处理函数，用于一旦出现 SIGCHLD 信号，就调用该函数
*********************************************/
#include        "unp.h"

void
sig_chld(int signo)
{
        pid_t   pid;
        int             stat;

        pid = wait(&stat);              // 等待子进程退出，返回的值是退出进程的 id，stat 用于保存被收集进程退出时的一些状态
        printf("child %d terminated\n", pid);
        return;
}
```

2. TCP 的三路握手：



3. TCP 的四次分手：

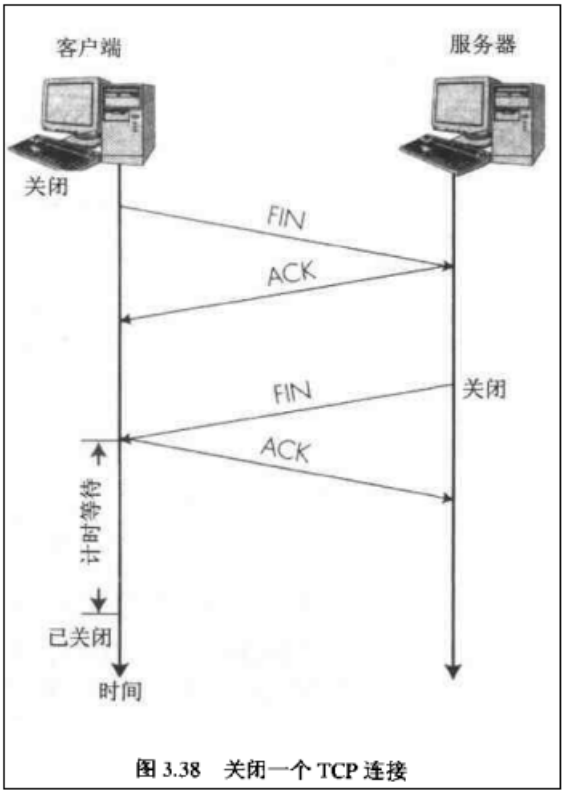


图 3.38　关闭一个 TCP 连接

4. 在 TCP 层，有个 FLAGS 字段：

- SYN 表示建立连接，
- FIN 表示关闭连接，
- ACK 表示响应，
- PSH 表示有 DATA 数据传输，
- RST 表示连接重置。

5. 一般地，当出现 FIN 包或 RST 包时，我们便认为客户端与服务器端断开了连接；而当出现 SYN 和 SYN＋ACK 包时，我们认为客户端与服务器建立了一个连接。

6. 服务器关闭一个连接后，如果 client 端由于写阻塞或读阻塞等原因没有及时处理 FIN 信号，client 端接着发数据，根据 TCP 协议的规定，会收到一个 RST 响应。若 client 再次往这个服务器发送数据时，系统会发出一个 SIGPIPE 信号给进程，告诉进程这个

连接已经断开了，不要再写了。该信号的缺省处理方法是退出进程。

7. 如果使用了多个套接字，那 SIGPIPE 信号无法告诉我们到底是哪个写出了错。
8. 当已连接的服务器突然崩溃时，客户端会不断地重传数据，持续重传 12 次，时间跨度大约为 9 分钟。为了避免耗时太长，需要给 write 等调用设置一个超时。
9. 当套接字之间发送二进制数据时，需要考虑不同的系统之间的数据类型的位数、大小端字节序的问题。
10. 因为根据 TCP 协议，收到对方的 FIN 包只意味着对方不会再发送任何消息。在一个双方正常关闭的流程中，收到 FIN 包的一端将剩余数据发送给对面（通过一次或多次 write），然后关闭 socket。
11. 假设 A 机器上的一个进程 a 正在和 B 机器上的进程 b 通信

- 假如 b 进程是异常终止的，发送 FIN 包是 OS 代劳的，b 进程已经不复存在，当机器再次收到该 socket 的消息时，会回应 RST。a 进程再次调用 write 时，B 机器的操作系统会给 a 进程发送 SIGPIPE，默认处理动作是终止进程。
- 不同于 b 进程退出（此时 OS 会负责为所有打开的 socket 发送 FIN 包），当 B 机器的 OS 崩溃/主机断电/网络不可达时，a 进程根本不会收到 FIN 包作为连接终止的提示。如果 a 进程阻塞在 read 上，那么结果只能是永远的等待。
- 假如 B 机器恰好在某个时候恢复和 A 机器的通路，并收到 a 某个重传的包，因为不能识别，所以会返回一个 RST，此时 a 进程上阻塞的 read 调用会返回错误 ECONNREST。

12. TCP 连接状态：

(1) CLOSED：初始状态，表示没有任何连接。

(2) LISTEN：Server 端的某个 Socket 正在监听来自远方的 TCP 端口的连接请求。

(3) SYN_SENT：发送连接请求后等待确认信息。当客户端 Socket 进行 Connect 连接时，会首先发送 SYN 包，随即进入 SYN_SENT 状态，然后等待 Server 端发送三次握手中的第 2 个包。

(4) SYN_RECEIVED：收到一个连接请求后回送确认信息和对等的连接请求，然后等待确认信息。通常是建立 TCP 连接的三次握手过程中的一个中间状态，表示 Server 端的 Socket 接收到来自 Client 的 SYN 包，并作出回应。

(5) ESTABLISHED：表示连接已经建立，可以进行数据传输。

(6) FIN_WAIT_1：主动关闭连接的一方等待对方返回 ACK 包。若 Socket 在 ESTABLISHED 状态下主动关闭连接并向对方发送 FIN 包，则进入 FIN_WAIT_1 状态，等待对方返回 ACK 包，此后还能读取数据，但不能发送数据。在正常情况下，无论对方处于何种状态，都应该马上返回 ACK 包，所以 FIN_WAIT_1 状态一般很难见到。

(7) FIN_WAIT_2：主动关闭连接的一方收到对方返回的 ACK 包后，等待对方发送 FIN 包。处于 FIN_WAIT_1 状态下的 Socket 收到了对方返回的 ACK 包后，便进入 FIN_WAIT_2 状态。由于 FIN_WAIT_2 状态下的 Socket 需要等待对方发送的 FIN 包，所有常常可以看到。若在 FIN_WAIT_1 状态下收到对方发送的同时带有 FIN 和 ACK 的包时，则直接进入 TIME_WAIT 状态，无须经过 FIN_WAIT_2 状态。

(8) TIME_WAIT：主动关闭连接的一方收到对方发送的 FIN 包后返回 ACK 包，然后等待足够长的时间以确保对方接收到 ACK，最后回到 CLOSED 状态，释放网络资源。

(9) CLOSE_WAIT：表示被动关闭连接的一方在等待关闭连接。当收到对方发送的 FIN 包后，相应的返回 ACK 包，然后进入 CLOSE_WAIT 状态。在该状态下，若己方还有数据未发送，则可以继续向对方进行发送，但不能再读取数据，直到数据发送完毕。

(10) LAST_ACK：被动关闭连接的一方在 CLOSE_WAIT 状态下完成数据的发送后便可向对方发送 FIN 包，然后等待对方返回 ACK 包。收到 ACK 包后便回到 CLOSED 状态，释放网络资源。

13. Select 和 poll 只需要指定描述符：对于 TCP 服务器而言，在使用 select 和 poll 前需要先进行 socket()、bind()、listen()，然后把 socket()得到的描述符加入 select()或 poll()中即可。

14. TCP 和 UDP 可以使用同一个端口，技术上每个协议的端口池是完全独立的。

```
listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
```

15. 为了增加可移植性，最好不使用协议相关的 gethostbyname()和 gethostbyaddr()，而改用协议无关的 getaddrinfo()和 getnameinfo()。

16. 相比传统的每个客户 fork 一次设计示范，预先创建一个子进程池或一个线程池的设计示范能够把进程控制 CPU 时间降低 10 倍或以上。

17. 让所有子进程或线程自行调用 accept 通常比让父进程或主线程独自调用 accept 并把描述符传递给子进程或线程来得简单而快速。

JavaScript 高级程序设计

第 5 章 引用类型

1. 冒号用法：

(1) A ? B : C 三元操作符；

(2) switch case 语句中；

(3) 对象直接量；

2. 所有对象都具有 toLocaleString()、toString()和 valueOf()方法。其中，调用数组的 toString()和 valueOf()方法会返回相同的值，即由数组中每个值的字符串形式拼接而成的一个以逗号分隔的字符串。

3. toLocaleString()方法：可根据本地时间把 Date 对象转换为字符串，并返回结果。

   toString()方法：可把一个逻辑值转换为字符串，并返回结果。

   valueOf()方法：可返回 Boolean 对象的原始值。

4. reduce() 方法：接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

```
<button onclick="myFunction()">点我</button>

<p>数组元素之和: <span id="demo"></span></p>

<script>
var numbers = [15.5, 2.3, 1.1, 4.7];

function getSum(total, num) {
    return total + Math.round(num);
}
function myFunction(item) {
    document.getElementById("demo").innerHTML = numbers.reduce(getSum, 0);
}
</script>
```

四舍五入后计算数组元素的总和：

5. reduceRight() 方法的功能和 reduce() 功能是一样的，不同的是 reduceRight()从数组的末尾向前将数组中的数组项做累加。

6. 函数内部的另一个特殊对象是 this，其行为与 Java 和 C#中的 this 大致类似。换句话说，this 引用的是函数据以执行的环境对象。

```
window.color = "red";
var o = { color: "blue" };
function sayColor(){
    alert(this.color);
}
sayColor(); //"red"
o.sayColor = sayColor;
o.sayColor(); //"blue"
```

由于在调用函数之前，this 的值并不确定，因此 this 可能会在代码执行过程中引用不同的对象。当在全局作用域中调用 sayColor()时，this 引用的是全局对象 window；换句

话说，对 this.color 求值会转换成对 window.color 求值，于是结果就返回了"red"。（调用之前、this 不确定、调用时、同级的 window 对象）

函数的名字仅仅是一个包含指针的变量而已。因此，即使是在不同的环境中执行，全局的 sayColor()函数与 o.sayColor()指向的仍然是同一个函数。（函数名字、包含指针的变量、sayColor()与 o.sayColor()、同一个函数）

7. 前面我们已经介绍了大多数内置对象，例如 Object、Array 和 String。ECMA-262 还定义了两个单体内置对象：Global 和 Math。
8. 不属于任何其他对象的属性和方法，最终都是 Global 对象的属性和方法。诸如 isNaN()、isFinite()、parseInt()以及 parseFloat()，实际上全都是 Global 对象的方法。（不属于任何对象、都属于）
9. Global 对象的 encodeURI()和 encodeURIComponent()方法可以对 URI 进行编码，以便发送给浏览器。（Global 对象、encodeURI()、encodeURIComponent()、编码 URI）

```
var uri = "http://www.wrox.com/illegal value.htm#start";

//"http://www.wrox.com/illegal%20value.htm#start"
alert(encodeURI(uri));

//"http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start"
alert(encodeURIComponent(uri));
```

使用 encodeURI()编码后的结果是除了空格之外的其他字符都原封不动，只有空格被替换成了%20。而 encodeURIComponent()方法则会使用对应的编码替换所有非字母数字字符。（encodeURI()、编码空格、encodeURIComponent()、除字母数字、都编码）

10. Global 对象的 eval()方法函数可计算某个字符串，并执行其中的的 JavaScript 代码。（eval()、执行代码）

```
eval("alert('hi')");
```

- 通过 eval()执行的代码被认为是包含该次调用的执行环境的一部分，因此被执行的代码具有与该执行环境相同的作用域链。这意味着通过 eval()执行的代码可以引用在包含环境中定义的变量，举个例子：（相同作用链、引用环境中的变量）

```
var msg = "hello world";
eval("alert(msg)"); //"hello world"
```

同样地，我们也可以在 eval()调用中定义一个函数，然后再在该调用的外部代码中引用这个函数：（eval()中定义函数、外部引用）

```
eval("function sayHi() { alert('hi'); }");
sayHi();
```

11. ECMAScript 虽然没有指出如何直接访问 Global 对象，但 Web 浏览器都是将这个全局对象作为 window 对象的一部分加以实现的。因此，在全局作用域中声明的所有变量和函数，就都成为了 window 对象的属性。（全局作用域、所有变量函数、window 对象的属性）

```
var color = "red";
```

```
function sayColor(){
 alert(window.color);
}

window.sayColor(); //"red"
```

12. 另一种取得Global对象的方法是使用以下代码：

```
var global = function(){
 return this;
}();
```

13. ECMAScript还为保存数学公式和信息提供了一个公共位置，即Math对象。与我们在JavaScript直接编写的计算功能相比，Math对象提供的计算功能执行起来要快得多。Math对象中还提供了辅助完成这些计算的属性和方法。（数学公式、Math对象）

(1) min()和max()方法用于确定一组数值中的最小值和最大值。（min()、最小、max()、最大）

```
var max = Math.max(3, 54, 32, 16);
alert(max); //54

var min = Math.min(3, 54, 32, 16);
alert(min); //3
```

(2) 将小数值舍入为整数的几个方法：Math.ceil()、Math.floor()和Math.round()。

- Math.ceil()执行向上舍入，即它总是将数值向上舍入为最接近的整数；（Math.ceil()、向上舍入）
- Math.floor()执行向下舍入，即它总是将数值向下舍入为最接近的整数；（Math.floor()、向下舍入）
- Math.round()执行标准舍入，即它总是将数值四舍五入为最接近的整数。（Math.round()、标准舍入、四舍五入）

```
alert(Math.ceil(25.9)); //26
alert(Math.ceil(25.5)); //26
alert(Math.ceil(25.1)); //26

alert(Math.round(25.9)); //26
alert(Math.round(25.5)); //26
alert(Math.round(25.1)); //25

alert(Math.floor(25.9)); //25
```

```
alert(Math.floor(25.5)); //25
alert(Math.floor(25.1)); //25
```

(3) Math.random()方法返回介于 0 和 1 之间一个随机数，不包括 0 和 1。套用下面的公式，就可以利用 Math.random()从某个整数范围内随机选择一个值。（Math.random()、0、1、随机数）

```
值 = Math.floor(Math.random() * 可能值的总数 + 第一个可能的值)
```

举例来说，如果你想选择一个 1 到 10 之间的数值，可以像下面这样编写代码：（0 到 1、Math.floor(Math.random() * 10 + 1)）

```
var num = Math.floor(Math.random() * 10 + 1);
```

第 6 章 面向对象的程序设计

1. ECMA-262 把对象定义为：“无序属性的集合，其属性可以包含基本值、对象或者函数。”严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。
2. 每个对象都是基于一个引用类型创建的。（对象、基于引用类型）
3. 创建自定义对象的最简单方式就是创建一个 Object 的实例，然后再为它添加属性和方法，如下所示。（创建对象、new Object）

```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";

person.sayName = function(){
 alert(this.name);
};
```

也可以使用对象字面量成为创建这种对象：（对象字面量)

```
var person = {
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",

 sayName: function(){
 alert(this.name);
 }
};
```

4. 为了表示特性是内部值，规范把它们放在了两对方括号中，例如[[Enumerable]]。（内部特性、两对方括号）
5. ECMAScript 中有两种属性：数据属性和访问器属性。（数据属性、访问器属性）

6. 数据属性包含一个数据值的位置。在这个位置可以读取和写入值。数据属性有 4 个描述其行为的特性。（数据属性、数据值）

- [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。这个特性默认值为 true。
- [[Enumerable]]：表示能否通过 for-in 循环返回属性。这个特性默认值为 true。（[[Enumerable]]、能否、通过 for-in 循环返回）
- [[Writable]]：表示能否修改属性的值。这个特性默认值为 true。（[[Writable]]、能否修改）
- [[Value]]：包含这个属性的数据值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。这个特性的默认值为 undefined。（[[Value]]、属性值）

```
var person = {
 name: "Nicholas"
};
```

这里创建了一个名为 name 的属性，为它指定的值是"Nicholas"。也就是说，[[Value]]特性将被设置为"Nicholas"，而对这个值的任何修改都将反映在这个位置。（name 属性、[[Value]]值、被设置为"Nicholas"）

7. 要修改属性默认的特性，必须使用 ECMAScript 5 的 Object.defineProperty()方法。（Object.defineProperty()、修改默认值）

```
var person = {};
Object.defineProperty(person, "name", {
 writable: false,
 value: "Nicholas"
});

alert(person.name); //"Nicholas"
person.name = "Greg";
alert(person.name); //"Nicholas"
```

这个例子创建了一个名为 name 的属性，它的值"Nicholas"是只读的。这个属性的值是不可修改的，如果尝试为它指定新值，则在非严格模式下，赋值操作将被忽略；在严格模式下，赋值操作将会导致抛出错误。

8. 在调用 Object.defineProperty()方法时，如果不指定，configurable、enumerable 和 writable 特性的默认值都是 false。（Object.defineProperty()、默认值为 false）
9. 访问器属性不包含数据值；它们包含一对儿 getter 和 setter 函数，不过，这两个函数都不是必需的。在读取访问器属性时，会调用 getter 函数，这个函数负责返回有效的值；在写入访问器属性时，会调用 setter 函数并传入新值，这个函数负责决定如何处理数据。访问器属性有如下 4 个特性。（访问器属性、getter()、获取值、setter()、设置值）

- [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的

默认值为 true。

- [[Enumerable]]：表示能否通过 for-in 循环返回属性。对于直接在对象上定义的属性，这个特性的默认值为 true。
- [[Get]]：在读取属性时调用的函数。默认值为 undefined。
- [[Set]]：在写入属性时调用的函数。默认值为 undefined。

10. 访问器属性不能直接定义，必须使用 Object.defineProperty()来定义。请看下面的例子。（定义访问器属性、Object.defineProperty()）

```
var book = {
 _year: 2004,
 edition: 1
};

Object.defineProperty(book, "year", {
 get: function(){
 return this._year;
 },
 set: function(newValue){

 if (newValue > 2004) {
 this._year = newValue;
 this.edition += newValue - 2004;
 }
 }
});

book.year = 2005;
alert(book.edition); //2
```

定义访问器属性时，其属性值为 set()和 get()两个函数的定义。其中 get()没有参数，set()具有一个包含要分配的值的参数。（访问器属性、定义 set()、get()）

11. 访问器属性只指定 getter 意味着属性是不能写，尝试写入属性会被忽略。在严格模式下，尝试写入只指定了 getter 函数的属性会抛出错误。类似地，没有指定 setter 函数的属性也不能读，否则在非严格模式下会返回 undefined，而在严格模式下会抛出错误。（只指定 getter、不能写、只指定 setter、不能读）

12. ECMAScript 5 又定义了一个 Object.defineProperties()方法。利用这个方法可以通过描述符一次定义多个属性。（Object.defineProperties()、定义多个属性）

```
var book = {};

Object.defineProperties(book, {
     _year: {
     value: 2004
     },
```

```
    edition: {
    value: 1
    },

    year: {
         get: function(){
         return this._year;
         },

         set: function(newValue){
              if (newValue > 2004) {
                   this._year = newValue;
                   this.edition += newValue - 2004;
              }
          }
    }
});
```

以上代码在 book 对外上定义了两个数据属性（_year 和 edition）和一个访问器属性（year）。

13. 使用 ECMAScript 5 的 Object.getOwnPropertyDescriptor()方法，可以取得给定属性的描述符。（Object.getOwnPropertyDescriptor()、获取、给定属性的描述符）

14. 工厂模式：用函数来封装以特定接口创建对象的细节。（工厂模式、函数封装接口、创建对象）

```
function createPerson(name, age, job){
 var o = new Object();
 o.name = name;
 o.age = age;
 o.job = job;
 o.sayName = function(){
 alert(this.name);
 };
 return o;
}

var person1 = createPerson("Nicholas", 29, "Software Engineer");
var person2 = createPerson("Greg", 27, "Doctor");
```

15. 构造函数模式：创建自定义的构造函数，从而定义自定义对象类型的属性和方法。

```
function Person(name, age, job){
 this.name = name;
 this.age = age;
 this.job = job;
```

```
 this.sayName = function(){
 alert(this.name);
 };
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");
```

16. 任何函数，只要通过 new 操作符来调用，那它就可以作为构造函数；而任何函数，如果不通过 new 操作符来调用，那它跟普通函数也不会有什么两样。（构造函数、new 来调用）

17. 每个函数都有一个 prototype（原型）属性，这个属性是一个指针，指向原型。

    原型，也就类似于 C++中的基类。

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
 alert(this.name);
};

var person1 = new Person();
person1.sayName(); //"Nicholas"

var person2 = new Person();
person2.sayName(); //"Nicholas"

alert(person1.sayName == person2.sayName); //true
```

在此，我们将 sayName()方法和所有属性直接添加到了 Person 的 prototype 属性中，构造函数变成了空函数。

18. JavaScript 的原型实际上就是另一个类型的对象：

```
function SuperType(){
this.property = true;
} S
uperType.prototype.getSuperValue = function(){
return this.property;
};
function SubType(){
this.subproperty = false;
}
```

```
//继承了 SuperType
SubType.prototype = new SuperType();
SubType.prototype.getSubValue = function (){
return this.subproperty;
};
var instance = new SubType();
alert(instance.getSuperValue()); //true
```

19. 实例属性指的是在构造函数中定义的属性，属性和方法都是不一样的引用地址。例如：

```
function CreateObject(name,age){
      this.name=name;  //实例属性
      this.age=age;
      this.run=function(){   //实例方法
            return this.name + this.age;
      }
}
```

20. 理解原型和实例的关键在于，创建对象时，new 后面的是构造函数。对象的原型属性指的是 prototype 部分内容，而对象的实例属性指的是构造函数中的内容。
21. obj.hasOwnProperty()：判断属性是存在。注意，该函数忽略原型属性，只判断自身是否存在该属性。
22. 在 JavaScript 中, constructor 属性返回对象的构造函数。返回值是函数的引用，不是函数名。
23. 有两种方式使用 in 操作符：单独使用和在 for-in 循环中使用。

- 在单独使用时，in 操作符会用于判断属性是否存在对象中，存在时返回 true，无论该属性存在于实例中还是原型中。看一看下面的例子。

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
 alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true
```

```
person1.name = "Greg";
alert(person1.name); //"Greg" ——来自实例
alert(person1.hasOwnProperty("name")); //true
alert("name" in person1); //true

alert(person2.name); //"Nicholas" ——来自原型
alert(person2.hasOwnProperty("name")); //false
alert("name" in person2); //true

delete person1.name;
alert(person1.name); //"Nicholas" ——来自原型
alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true
```

in 操作符只要通过对象能够访问到属性就返回 true，hasOwnProperty()只在属性存在于实例中时才返回 true，因此只要 in 操作符返回 true 而 hasOwnProperty()返回 false，就可以确定属性是原型中的属性。

- for-in 语句用于对数组或者对象的属性进行循环操作。使用 for-in 循环时，返回的是所有能够通过对象访问的、可枚举的属性，其中既包括存在于实例中的属性，也包括存在于原型中的属性。

24. 前面例子中每添加一个属性和方法就要敲一遍 Person.prototype。为减少不必要的输入，也为了从视觉上更好地封装原型的功能，更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象，如下面的例子所示。

```
function Person(){
}

Person.prototype = {
 name : "Nicholas",
 age : 29,
 job: "Software Engineer",
 sayName : function () {
 alert(this.name);
 }
};
```

我们将 Person.prototype 设置为等于一个以对象字面量形式创建的新对象。最终结果相同，但有一个例外：constructor 属性变成了新对象的 constructor 属性（指向 Object 构造函数），不再指向 Person 函数。（constructor 属性、不再指向 Person）

```
var friend = new Person();

alert(friend instanceof Object); //true
alert(friend instanceof Person); //true
alert(friend.constructor == Person); //false
alert(friend.constructor == Object); //true
```

25. 构造函数：

```
function Person(name, age, job) {
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = function() { alert(this.name) }
}
var person1 = new Person('Zaxlct', 28, 'Software Engineer');
var person2 = new Person('Mick', 23, 'Doctor');
```

person1 和 person2 都是 Person 的实例。这两个实例都有一个 constructor （构造函数）属性，该属性（是一个指针）指向 Person。（constructor、指向 Person）

26. JavaScript 中的实例，指的类似于 C++中的对象。

27. 由于在原型中查找值的过程是一次搜索，因此即使是先创建了实例后修改原型，我们对原型对象所做的任何修改都能够立即从实例上反映出来。

```
var friend = new Person();

Person.prototype.sayHi = function(){
 alert("hi");
};

friend.sayHi(); //"hi"（没有问题！）
```

28. 动态原型模式：

```
function Person(name, age, job){

 //属性
    this.name = name;
    this.age = age;
    this.job = job;

    //方法
    if (typeof this.sayName != "function"){

        Person.prototype.sayName = function(){
            alert(this.name);
        };

    }
}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName();
```

如果原型中没有定义指定函数，则定义该函数，有该函数则不执行。

29. 使用动态原型模式时，不能使用对象字面量重写原型。如果在已经创建了实例的情况下重写原型，那么就会切断现有实例与新原型之间的联系。

30. 寄生构造函数模式的基本思想是创建一个函数，该函数的作用仅仅是封装创建对象的代码，然后再返回新创建的对象；但从表面上看，这个函数又很像是典型的构造函数。下面是一个例子。

```
function Person(name, age, job){
    var o = new Object();
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName(); //"Nicholas"
```

31. 所谓稳妥对象，指的是没有公共属性，而且其方法也不引用 this 的对象。

```
function Person(name, age, job){

    //创建要返回的对象
    var o = new Object();

    //可以在这里定义私有变量和函数

    //添加方法
    o.sayName = function(){
        alert(name);
    };

    //返回对象
    return o;
}
```

注意，在以这种模式创建的对象中，除了使用 sayName()方法之外，没有其他办法访问 name 的值。可以像下面使用稳妥的 Person 构造函数。

```
var friend = Person("Nicholas", 29, "Software Engineer");
friend.sayName(); //"Nicholas"
```

32. 当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查

不到，就去找原型的原型，一直找到最顶层为止。

```
function Person() {

}

Person.prototype.name = 'Kevin';

var person = new Person();

person.name = 'Daisy';
console.log(person.name) // Daisy

delete person.name;
console.log(person.name) // Kevin
```

33. ECMAScript 只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

34. 原型链：对象 A 的原型是 B，B 的原型是 C，C 的原型是 D，D 的原型最后是 Null。也就是类似 C++的 A 继承自 B，B 又继承自 C。（原型链、类似 C++的继承链）

35. 继承是通过创建 SuperType 的实例，并将该实例赋给 SubType.prototype 实现的。实现的本质是重写原型对象，代之以一个新类型的实例。

```
function SuperType(){
    this.property = true;
}
SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType(){
    this.subproperty = false;
}

//继承了 SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

var instance = new SubType();
alert(instance.getSuperValue()); //true
```

instance 指向 SubType 的原型，SubType 的原型又指向 SuperType 的原型。

36. 所有函数的默认原型都是 Object 的实例，因此默认原型都会包含一个内部指针，指向

Object.prototype。

37. 通过两种方式来确定原型和实例之间的关系。

- 第一种方式是使用 instanceof 操作符，只要用这个操作符来测试实例与原型链中出现过的构造函数，结果就会返回 true。以下几行代码就说明了这一点。

```
alert(instance instanceof Object); //true
alert(instance instanceof SuperType); //true
alert(instance instanceof SubType); //true
```

- 第二种方式是使用 isPrototypeOf()方法。

```
alert(Object.prototype.isPrototypeOf(instance)); //true
alert(SuperType.prototype.isPrototypeOf(instance)); //true
alert(SubType.prototype.isPrototypeOf(instance)); //true
```

38. 给原型添加方法的代码一定要放在替换原型的语句之后。

```
function SuperType(){
 this.property = true;
}

SuperType.prototype.getSuperValue = function(){
 return this.property;
};

function SubType(){
 this.subproperty = false;
}

//继承了 SuperType
SubType.prototype = new SuperType();

//添加新方法
SubType.prototype.getSubValue = function (){
 return this.subproperty;
};

//重写超类型中的方法
SubType.prototype.getSuperValue = function (){
 return false;
};

var instance = new SubType();
alert(instance.getSuperValue()); //false
```

39. 通过原型链实现继承时，不能使用对象字面量创建原型方法。因为这样做就会重写原型链。

40. 原型链的问题：在通过原型来实现继承时，子类的原型实际上会变成父类原型的实例。下列代码可以用来说明这个问题。

```
function SuperType(){
 this.colors = ["red", "blue", "green"];
}

function SubType(){
}

//继承了 SuperType
SubType.prototype = new SuperType();

var instance1 = new SubType();
instance1.colors.push("black");
alert(instance1.colors); //"red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors); //"red,blue,green,black"
```

当 SubType 通过原型链继承了 SuperType 之后，SubType.prototype 就变成了 SuperType 的一个实例，因此它也拥有了一个它自己的 colors 属性。结果是 SubType 的所有实例都会共享这一个 colors 属性。

41. 借用构造函数的基本思想相当简单，即在子类型构造函数的内部调用超类型构造函数。通过使用 apply()和 call()方法也可以在新创建的对象上执行构造函数，如下所示：

```
function SuperType(){
 this.colors = ["red", "blue", "green"];
}

function SubType(){
 //继承了 SuperType
 SuperType.call(this);
}

var instance1 = new SubType();
instance1.colors.push("black");
alert(instance1.colors); //"red,blue,green,black"

var instance2 = new SubType();
alert(instance2.colors); //"red,blue,green"
```

这样，SubType 的每个实例就都会具有自己的 colors 属性的副本了。

42. 相对于原型链而言，借用构造函数有一个很大的优势，即可以在子类型构造函数中向

超类型构造函数传递参数。看下面这个例子。（借用构造函数、向超类构造函数传参数）

```
function SuperType(name){
 this.name = name;
}

function SubType(){
 //继承了 SuperType，同时还传递了参数
 SuperType.call(this, "Nicholas");

 //实例属性
 this.age = 29;
}

var instance = new SubType();
alert(instance.name); //"Nicholas";
alert(instance.age); //29
```

43. 组合继承，使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

```
<script>
    function Parent(age){
        this.name = ['mike','jack','smith'];
        this.age = age;
    }
    Parent.prototype.run = function () {
        return this.name  + ' are both' + this.age;
    };
    function Child(age){
        Parent.call(this,age);//对象冒充，给超类型传参
    }
    Child.prototype = new Parent();//原型链继承
    var test = new Child(21);//写 new Parent(21)也行
    alert(test.run());//mike,jack,smith are both21
</script>
```

44. 寄生式继承：也就是一个对象的定义和赋值都在一个函数里实现。以下代码示范了寄生式继承模式。

```
function createAnother(original){
 var clone = object(original); //通过调用函数创建一个新对象
 clone.sayHi = function(){ //以某种方式来增强这个对象
 alert("hi");
```

```
 };
 return clone; //返回这个对象
}
```

createAnother()函数接收了一个参数，也就是将要作为新对象基础的对象。然后，把这个对象（original）传递给 object()函数，将返回的结果赋值给 clone。再为 clone 对象添加一个新方法 sayHi()，最后返回 clone 对象。可以像下面这样来使用 createAnother()函数：

```
var person = {
 name: "Nicholas",
 friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = createAnother(person);
anotherPerson.sayHi(); //"hi"
```

45. 在主要考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。

46. 组合继承是 JavaScript 最常用的继承模式；不过，它也有自己的不足。组合继承最大的问题就是无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。

47. 所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。寄生组合式继承的基本模式如下所示。

```
function inheritPrototype(subType, superType){
 var prototype = object(superType.prototype); //创建超类型原型的一个副本
 prototype.constructor = subType; //为创建的副本添加 constructor 属性
 subType.prototype = prototype; //将新创建的对象赋值给子类型的原型
}
```

这个函数接收两个参数：子类型构造函数和超类型构造函数。

这样，我们就可以用调用 inheritPrototype()函数的语句，去替换前面例子中为子类型原型赋值的语句了，例如：

```
function SuperType(name){
 this.name = name;
 this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function(){
 alert(this.name);
};

function SubType(name, age){
 SuperType.call(this, name);

 this.age = age;
```

```
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function(){
 alert(this.age);
};
```

这里要注意，调用 inheritPrototype 前，要先定义子类。

第 7 章 函数表达式

1. 定义函数的方式有两种：一种是函数声明，另一种就是函数表达式。

(1) 函数声明的语法是这样的。

```
function functionName(arg0, arg1, arg2) {
 //函数体
}
```

(2) 函数表达式有几种不同的语法形式。下面是最常见的一种形式。

```
var functionName = function(arg0, arg1, arg2){
 //函数体
};
```

这种情况下创建的函数叫做匿名函数，因为 function 关键字后面没有标识符。（匿名函数）

2. 函数声明的一个重要特征就是函数声明提升，意思是在执行代码之前会先读取函数声明，变量（函数）的声明会被提升到作用域的最前面，即使写代码的时候是写在最后面，也还是会被提升至最前面。这就意味着可以把函数声明放在调用它的语句后面。

```
sayHi();
function sayHi(){
    alert("Hi!");
}
```

这个例子不会抛出错误，因为在代码执行之前会先读取函数声明。

3. 函数表达式与其他表达式一样，在使用前必须先赋值。以下代码会导致错误。（函数表达式、先赋值）

```
sayHi(); //错误：函数还不存在
var sayHi = function(){
    alert("Hi!");
};
```

4. 函数声明在 JS 解析时进行函数提升，因此在同一个作用域内，不管函数声明在哪里定义，该函数都可以进行调用。而函数表达式的值是在 JS 运行时确定，并且在表达式赋值完成后，该函数才能调用。

5. 函数声明与函数表达式之间的区别：

```
//不要这样做!
if(condition){
    function sayHi(){
    alert("Hi!");
   }
} else {
    function sayHi(){
       alert("Yo!");
    }
}
```

这在 JavaScript 中是无效语法。而在函数表达式中却可以使用：

```
//可以这样做
var sayHi;

if(condition){
    sayHi = function(){
    alert("Hi!");
    };
} else {
    sayHi = function(){
    alert("Yo!");
    };
}
```

6. arguments.callee()：返回正被执行的 Function 对象，也就是所指定的 Function 对象的正文。

functionName.caller()：返回函数的调用者。如果函数是由 Javascript 程序的顶层调用的，那么 caller 包含的就是 null。

```
function factorial(num){
    if (num <= 1){
    return 1;
} else {
    return num * arguments.callee(num-1);
    }
}
```

在编写递归函数时，使用 arguments.callee 总比使用函数名更保险。

7. 在严格模式下，不能通过脚本访问 arguments.callee，访问这个属性会导致错误。不过，可以使用命名函数表达式来进行递归调用。

```
var factorial = (function f(num){
 if (num <= 1){
 return 1;
```

```
 } else {
 return num * f(num-1);
 }
});
```

以上代码创建了一个名为 f()的命名函数表达式，然后将它赋值给变量 factorial。即便把函数赋值给了另一个变量，函数的名字 f 仍然有效，所以递归调用照样能正确完成。这种方式在严格模式和非严格模式下都行得通。

8. 闭包是指有权访问另一个函数作用域内部的变量的函数。创建闭包的常见方式，就是在一个函数内部创建另一个函数，然后将该作为外层函数的返回值：

```
function f1(){
    var n=999;
    function f2(){
        alert(n);
    }
    return f2;
}
var result=f1();
result(); // 999
```

f2 就是个闭包。

9. 闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

10. 闭包的作用就是在 outer 执行完毕并返回后，闭包使 javascript 的垃圾回收机制不会回收 outer 所占的内存：

```
function outer() {
    var i = 100;
    function inner() {
         console.log(i++);
    }
    return inner;
}
var rs = outer();
rs();   //100
rs();   //101
rs();   //102
```

11. 由于作用域链的机制，闭包只能取得包含函数中任何变量的最后一个值。

```
function f() {
    var rs = [];

    for (var i=0; i <10; i++) {
        rs[i] = function() {
            return i;
```

```
        };
    }

    return rs;
}

var fn = f();

for (var i = 0; i < fn.length; i++) {
    console.log('函数fn[' + i + ']()返回值:' + fn[i]());
}
```

函数会返回一个数组，表面上看，似乎每个函数都应该返回自己的索引值，实际上，每个函数都返回 10，这是因为函数也是对象，是一个引用，所以 rs 是引用，引用的都是同一变量 i。

我们可以通过创建另一个匿名函数来进行参数传值，强制让闭包的行为符合预期。

```
function f() {
    var rs = [];

    for (var i=0; i <10; i++) {
        rs[i] = function(num) {
            return function() {
                return num;
            };
        }(i);
    }

    return rs;
}

var fn = f();

for (var i = 0; i < fn.length; i++) {
    console.log('函数fn[' + i + ']()返回值:' + fn[i]());
}
```

这个版本中，我们没有直接将闭包赋值给数组，而是定义了一个匿名函数，并将立即执行匿名函数的结果赋值给数组。这里匿名函数有一个参数 num，在调用每个函数时，我们传入变量 i，由于参数是按值传递的，所以就会将变量 i 复制给参数 num。而在这个匿名函数内部，又创建了并返回了一个访问 num 的闭包，这样，rs 数组中每个函数都有自己 num 变量的一个副本，因此就可以返回不同的数值了。

12. 匿名函数的执行环境具有全局性，因此其 this 对象通常指向 window。

13. JavaScript 没有块级作用域的概念。这意味着在块语句中定义的变量，实际上是在包含函数中而非语句中创建的，来看下面的例子。（

```
function outputNumbers(count){
 for (var i=0; i < count; i++){
 alert(i);
 }
 alert(i); //计数
}
```

在 c++中，i 只在 for 语句中有效。而在 JavaScript 中，i 在整个函数有效。

14. 如果出现重复声明变量，则会忽略后面声明的变量，而不会报错。

15. 匿名函数可以用来模仿块级作用域：

```
(function(){
 //这里是块级作用域
})();
```

16. 有权访问私有变量和私有函数的公有方法称为特权方法。有两种在对象上创建特权方法的方式。

```
function MyObject(){

 //私有变量和私有函数
 var privateVariable = 10;

 function privateFunction(){
 return false;
 }

 //特权方法
 this.publicMethod = function (){
 privateVariable++;
 return privateFunction();
 };
}
```

17. 模块模式使用了 JavaScript 中的闭包用来给你方法中的隐私一些控制数据或方法，第三方应用程序不能访问私有数据或覆盖它。

```
var singleton = function(){

    //私有变量和私有函数
    var privateVariable = 10;

    function privateFunction(){
    return false;
}
```

```
 //特权/公有方法和属性
    return {

        publicProperty: true,

        publicMethod : function(){
           privateVariable++;
        return privateFunction();
    }

 };
}();
```

18. 增强的模块模式，即在返回对象之前加入对其增强的代码。这种增强的模块模式适合那些单例必须是某种类型的实例，同时还必须添加某些属性和（或）方法对其加以增强的情况。来看下面的例子。

```
var single = function(){
  var name = 'yeye';
  var age = 26;
  var app = new Array();
  app.name = name;
  app.setName = function(val){
      name = val;
      age++;
      return [name,age]
      }
  return app
 }()
single instanceof Array
```

在对象返回前，我们创建了一个数组实例,并为这个实例添加了属性和方法。这就是增强的模块模式了。

第 8 章 BOM

1. BOM 的核心对象是 window，它表示浏览器的一个实例。
2. 在网页中定义的任何一个对象、变量和函数，都以 window 作为其 Global 对象，因此有权访问 parseInt()等方法。
3. 所有在全局作用域中声明的变量、函数都会变成 window 对象的属性和方法。
4. 定义全局变量与在 window 对象上直接定义属性还是有一点差别：全局变量不能通过 delete 操作符删除，而直接在 window 对象上的定义的属性可以。
5. 使用 var 语句添加的 window 属性有一个名为[[Configurable]]的特性，这个特性的值如果被设置为 false，那这个属性不可以通过 delete 操作符删除。

6. 尝试访问未声明的变量会抛出错误，但是如果指定为 window 对象，则不会抛出错误。

```
//这里会抛出错误，因为 oldValue 未定义
var newValue = oldValue;

//这里不会抛出错误，因为这是一次属性查询
//newValue 的值是 undefined
var newValue = window.oldValue;
```

7. 如果页面中包含框架，则每个框架都拥有自己的 window 对象，并且保存在 frames 集合中。在 frames 集合中，可以通过数值索引（从 0 开始，从左至右，从上到下）或者框架名称来访问相应的 window 对象。每个 window 对象都有一个 name 属性，其中包含框架的名称。下面是一个包含框架的页面：

```
<html>
 <head>
 <title>Frameset Example</title>
 </head>
 <frameset rows="160,*">
 <frame src="frame.htm" name="topFrame">
 <frameset cols="50%,50%">
 <frame src="anotherframe.htm" name="leftFrame">
 <frame src="yetanotherframe.htm" name="rightFrame">
 </frameset>
 </frameset>
</html>
```

以上代码创建了一个框架集，其中一个框架居上，两个框架居下。对这个例子而言，可以通过 window.frames[0]或者 window.frames["topFrame"]来引用上方的框架。不过，恐怕你最好使用 top 而非 window 来引用这些框架，例如，通过 top.frames[0]。

8. top 对象始终指向最高（最外）层的框架，也就是浏览器窗口。

9. 下图是访问框架的方式：

10. 与 top 相对的另一个 window 对象是 parent。顾名思义，parent 对象始终指向当前框架的直接上层框架。在某些情况下，parent 有可能等于 top；但在没有框架的情况下，parent 一定等于 top，此时它们都等于 window。
11. 除非最高层窗口是通过 window.open()打开的，否则其 window 对象的 name 属性不会包含任何值。
12. 与框架有关的最后一个对象是 self，它始终指向 window；实际上，self 和 window 对象可以互换使用。引入 self 对象的目的只是为了与 top 和 parent 对象对应起来，因此它不格外包含其他值。
13. 在使用框架的情况下，浏览器中不同框架下的 Global 对象所指的内容不同。在每个框架中定义的全局变量会自动成为框架中 window 对象的属性。
14. 用来确定和修改 window 对象位置的属性和方法有很多。IE、Safari、Opera 和 Chrome 都提供了 screenLeft 和 screenTop 属性，分别用于表示窗口相对于屏幕左边和上边的位置。Firefox 则在 screenX 和 screenY 属性中提供相同的窗口位置信息，Safari 和 Chrome 也同时支持这两个属性。使用下列代码可以跨浏览器取得窗口左边和上边的位置。

```
var leftPos = (typeof window.screenLeft == "number") ?
window.screenLeft : window.screenX;
var topPos = (typeof window.screenTop == "number") ?
window.screenTop : window.screenY;
```

15. 在 Opera 和 Chrome 中，screenLeft 和 screenTop 中保存的是从屏幕左边和上边到由 window 对象表示的页面可见区域的距离。

但是，在 Firefox 和 Safari 中，screenY 或 screenTop 中保存的是整个浏览器窗口相对于屏幕的坐标值，即在窗口的 y 轴坐标为 0 时返回 0。

16. 无法在跨浏览器的条件下取得窗口左边和上边的精确坐标值。然而，使用 moveTo()和 moveBy()方法倒是有可能将窗口精确地移动到一个新位置。

```
//将窗口移动到屏幕左上角
window.moveTo(0,0);

//将窗向下移动 100 像素
window.moveBy(0,100);

//将窗口移动到(200,300)
window.moveTo(200,300);

//将窗口向左移动 50 像素
window.moveBy(-50,0);
```

需要注意的是，这两个方法可能会被浏览器禁用。另外，这两个方法都不适用于框架，只能对最外层的 window 对象使用。

17. 跨浏览器确定一个窗口的大小不是一件简单的事。Firefox、Safari、Opera 和 Chrome 均为此提供了 4 个属性：innerWidth、innerHeight、outerWidth 和 outerHeight。对于不同浏览器，这 4 个属性所指的内容有所一同。由于与桌面浏览器间存在这些差异，最好是先检测一下用户是否在使用移动设备，然后再决定使用哪个属性。

18. 使用 resizeTo()和 resizeBy()方法可以调整浏览器窗口的大小。（

19. 使用 window.open()方法既可以导航到一个特定的 URL，也可以打开一个新的浏览器窗口。函数返回一个指向新窗口的引用。

20. 调用 close()方法还可以关闭新打开的窗口。

```
var wroxWin = window.open("http://www.wrox.com/","wroxWindow",
 "height=400,width=400,top=10,left=10,resizable=yes");

//调整大小
wroxWin.resizeTo(500,500);

//移动位置
wroxWin.moveTo(100,100);

wroxWin.close();
```

这个方法仅适用于通过 window.open()打开的弹出窗口。对于浏览器的主窗口，如果没有得到用户的允许是不能关闭它的。弹出窗口倒是可以调用 top.close()在不经用户允许的情况下关闭自己。

21. 新创建的 window 对象有一个 opener 属性，其中保存着打开它的原始窗口对象。这个属性只在弹出窗口中的最外层 window 对象（top）中有定义，而且指向调用 window.open()的窗口或框架。

```
var wroxWin = window.open("http://www.wrox.com/","wroxWindow",
"height=400,width=400,top=10,left=10,resizable=yes");

alert(wroxWin.opener == window); //true
```

22. JavaScript 允许通过设置超时值和间歇时间值来调度代码在特定的时刻执行。

- 超时调用需要使用 window 对象的 setTimeout()方法。

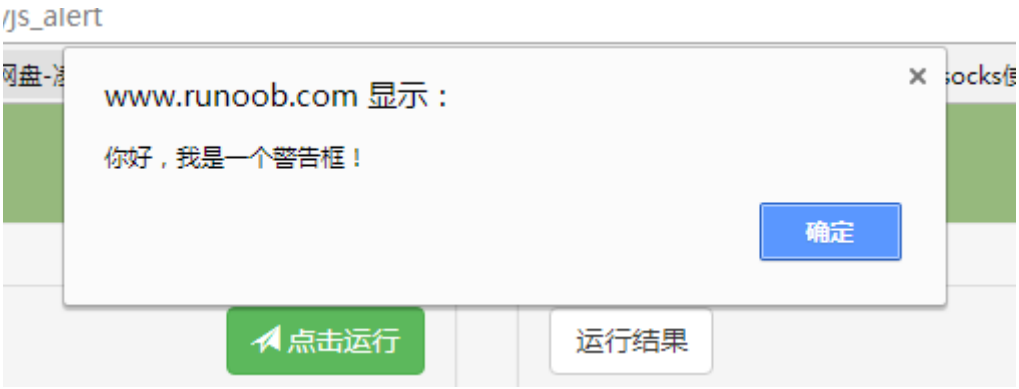
```
//不建议传递字符串！
setTimeout("alert('Hello world!') ", 1000);

//推荐的调用方式
setTimeout(function() {
 alert("Hello world!");
}, 1000);
```

- 间歇调用需要使用 setInterval()。

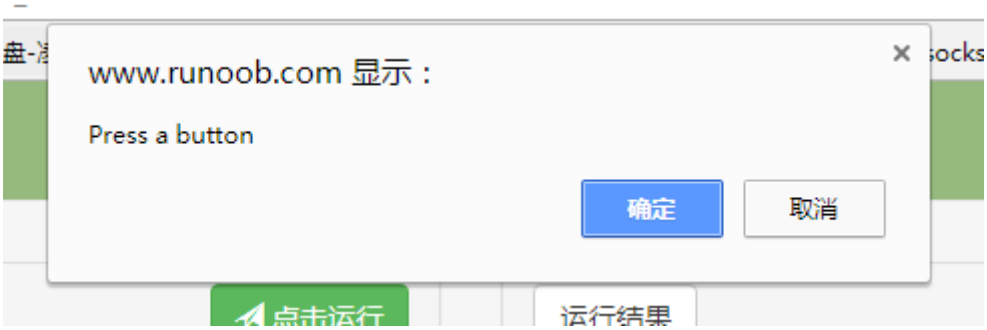
```
//不建议传递字符串！
setInterval ("alert('Hello world!') ", 10000);

//推荐的调用方式
setInterval (function() {
 alert("Hello world!");
}, 10000);
```

23. 浏览器通过 alert()、confirm()和 prompt()方法可以调用系统对话框向用户显示消息。系统对话框与在浏览器中显示的网页没有关系，也不包含 HTML。

- alert() 方法用于显示带有一条指定消息和一个 OK 按钮的警告框。（alter()、警告框）



- confirm() 方法用于显示一个带有指定消息和 OK 及取消按钮的对话框。（confirm()、带 OK 和取消按钮）



- pro mpt() 方法用于显示可提示用户进行输入的对话框。（prompt、提示用户输入）

24. 还有两个可以通过 JavaScript 打开的对话框，即“查找”和“打印”。（查找、find()、打印、print()）

```
//显示“打印”对话框
window.print();

//显示“查找”对话框
window.find();
```

25. document 对象是文档的根节点，window.document 属性就指向这个对象。也就是说，只要浏览器开始载入 HTML 文档，这个对象就开始存在了，可以直接调用。

26. Location 对象包含有关当前 URL 的信息。事实上，location 对象是很特别的一个对象，因为它既是 window 对象的属性，也是 document 对象的属性；换句话说，window.location 和 document.location 引用的是同一个对象。（location、当前 URL、既是 window 的属性、又是 document 的属性）

27. location 对象将 URL 解析为独立的片段，让开发人员可以通过不同的属性访问这些片段。（location 对象、URL、独立的片段）

属性	描述	说明
hash	设置或返回从井号 (#) 开始的 URL（锚）。	"#contents"
host	设置或返回主机名和当前 URL 的端口号。	"www.wrox.com:80"
hostname	设置或返回当前 URL 的主机名。	"www.wrox.com"

href	设置或返回完整的 URL。	"http:/www.wrox.com"
pathname	设置或返回当前 URL 的路径部分。	"/WileyCDA/"
port	设置或返回当前 URL 的端口号。	"8080"
protocol	设置或返回当前 URL 的协议。	"http:"
search	设置或返回从问号 (?) 开始的 URL。	"?q=javascript"

28. location.search 返回从问号到 URL 末尾的所有内容。在这些内容中，每个&分隔的是一个查询字符串参数。（search、问号到末尾、&分隔查询字符串参数）

29. assign() 方法可加载一个新的文档。该方法和下面两条语句实现同样的效果：（assign()、加载新文档）

```
window.location = "http://www.wrox.com";
location.href = "http://www.wrox.com";
```

30. location.reload()：重新加载当前文档。

31. location.replace()：用新的文档替换当前文档。

32. Navigator 对象包含有关浏览器的信息。Navigator 对象属性如下：（Navigator、浏览器有关的信息）

属性	描述
appCodeName	返回浏览器的代码名。
appMinorVersion	返回浏览器的次级版本。
appName	返回浏览器的名称。
appVersion	返回浏览器的平台和版本信息。
browserLanguage	返回当前浏览器的语言。
cookieEnabled	返回指明浏览器中是否启用 cookie 的布尔值。
cpuClass	返回浏览器系统的 CPU 等级。
onLine	返回指明系统是否处于脱机模式的布尔值。
platform	返回运行浏览器的操作系统平台。
systemLanguage	返回 OS 使用的默认语言。
userAgent	返回由客户机发送服务器的 user-agent 头部的值。
userLanguage	返回 OS 的自然语言设置。

33. 检测浏览器中是否安装了特定的插件是一种最常见的检测例程。对于非 IE 浏览器，可以使用 navigator.plugins 数组来达到这个目的。该数组中的每一项都包含下列属性。（检测插件、navigator.plugins 数组）

- name：插件的名字。
- description：插件的描述。
- filename：插件的文件名。
- length：插件所处理的 MIME 类型数量。

```
//检测插件（在 IE 中无效）
function hasPlugin(name){
 name = name.toLowerCase();
 for (var i=0; i < navigator.plugins.length; i++){
 if (navigator. plugins [i].name.toLowerCase().indexOf(name) > -1){
 return true;
 }
 }

 return false;
}

//检测 Flash
alert(hasPlugin("Flash"));

//检测 QuickTime
alert(hasPlugin("QuickTime"));
```

name 是要检测的插件名。

34. Screen 对象包含有关客户端显示屏幕的信息。Screen 对象属性如下：（Screen 对象、屏幕信息）

属性	描述
availHeight	返回显示屏幕的高度 (除 Windows 任务栏之外)。
availWidth	返回显示屏幕的宽度 (除 Windows 任务栏之外)。
bufferDepth	设置或返回调色板的比特深度。
colorDepth	返回目标设备或缓冲器上的调色板的比特深度。
deviceXDPI	返回显示屏幕的每英寸水平点数。
deviceYDPI	返回显示屏幕的每英寸垂直点数。
fontSmoothingEnabled	返回用户是否在显示控制面板中启用了字体平滑。
height	返回显示屏幕的高度。
logicalXDPI	返回显示屏幕每英寸的水平方向的常规点数。
logicalYDPI	返回显示屏幕每英寸的垂直方向的常规点数。
pixelDepth	返回显示屏幕的颜色分辨率（比特每像素）。
updateInterval	设置或返回屏幕的刷新率。
width	返回显示器屏幕的宽度。

35. History 对象包含用户（在浏览器窗口中）访问过的 URL。History 对象是 window 对象的一部分，可通过 window.history 属性对其进行访问。该对象只有一个 length 属性，该属性返回浏览器历史列表中的 URL 数量。（History 对象、访问过的 URL）

36. History 对象方法：

(1) back()：加载 history 列表中的前一个 URL。（back()、前一个 URL）

(2) forward()：加载 history 列表中的下一个 URL。（forward()、下一个 URL）

(3) go()：加载 history 列表中的某个具体页面。（go()、加载历史 URL）

37. Window 对象属性：

(1) closed：返回窗口是否已被关闭。

(2) defaultStatus：设置或返回窗口状态栏中的默认文本。

(3) document：对 Document 对象的只读引用。请参阅 Document 对象。

(4) history：对 History 对象的只读引用。请参数 History 对象。

(5) innerheight：返回窗口的文档显示区的高度。

(6) innerwidth：返回窗口的文档显示区的宽度。

(7) length：设置或返回窗口中的框架数量。

(8) location：用于窗口或框架的 Location 对象。请参阅 Location 对象。

(9) name：设置或返回窗口的名称。

(10) Navigator：对 Navigator 对象的只读引用。请参数 Navigator 对象。

(11) opener：返回对创建此窗口的窗口的引用。

(12) outerheight：返回窗口的外部高度。

(13) outerwidth：返回窗口的外部宽度。

(14) pageXOffset：设置或返回当前页面相对于窗口显示区左上角的 X 位置。

(15) pageYOffset：设置或返回当前页面相对于窗口显示区左上角的 Y 位置。

(16) parent：返回父窗口。

(17) Screen：对 Screen 对象的只读引用。请参数 Screen 对象。

(18) self：返回对当前窗口的引用。等价于 Window 属性。

(19) status：设置窗口状态栏的文本。

(20) top：返回最顶层的先辈窗口。

(21) window：window 属性等价于 self 属性，它包含了对窗口自身的引用。

(22) screenLeft、screenTop、screenX、screenY：只读整数。声明了窗口的左上角在屏幕上的的 x 坐标和 y 坐标。IE、Safari 和 Opera 支持 screenLeft 和 screenTop，而 Firefox 和 Safari 支持 screenX 和 screenY。

38. Window 对象方法：

(1) alert()：显示带有一段消息和一个确认按钮的警告框。

(2) blur()：把键盘焦点从顶层窗口移开。

(3) clearInterval()：取消由 setInterval() 设置的 timeout。

(4) clearTimeout()：取消由 setTimeout() 方法设置的 timeout。

(5) close()：关闭浏览器窗口。

(6) confirm()：显示带有一段消息以及确认按钮和取消按钮的对话框。

(7) createPopup()：创建一个 pop-up 窗口。

(8) focus()：把键盘焦点给予一个窗口。

(9) moveBy()：可相对窗口的当前坐标把它移动指定的像素。

(10) moveTo()：把窗口的左上角移动到一个指定的坐标。

(11) open()：打开一个新的浏览器窗口或查找一个已命名的窗口。

(12) print()：打印当前窗口的内容。

(13) prompt()：显示可提示用户输入的对话框。

(14) resizeBy()：按照指定的像素调整窗口的大小。

(15) resizeTo()：把窗口的大小调整到指定的宽度和高度。

(16) scrollBy()：按照指定的像素值来滚动内容。

(17) scrollTo()：把内容滚动到指定的坐标。

(18) setInterval()：按照指定的周期（以毫秒计）来调用函数或计算表达式。

(19) setTimeout()：在指定的毫秒数后调用函数或计算表达式。

第 9 章 客户端检测

1. 先设计最通用的方案，然后再使用特定于浏览器的技术增强该方案。（先通用、后特定）
2. 最常用也最为人们广泛接受的客户端检测形式是能力检测（又称特性检测）。能力检测的目标不是识别特定的浏览器，而是识别浏览器的能力。能力检测的基本模式如下：（能力检测）

```
if (object.propertyInQuestion){
 //使用 object.propertyInQuestion
}
```

举例如下：

```
function getElement(id){
 if (document.getElementById){
 return document.getElementById(id);
 } else if (document.all){
 return document.all[id];
 } else {
 throw new Error("No way to retrieve element!");
 }
}
```

getElement()函数的用途是返回具有给定 ID 的元素。因为 document.getElementById()是实现这一目的的标准方式，所以一开始就测试了这个方法。如果该函数存在（不是未定义），则使用该函数。否则，就要继续检测 document.all 是否存在，如果是，则使用它。如果上述两个特性都不存在（很有可能），则创建并抛出错误，表示这个函数无法使用。

3. 检测某个对象是否支持排序。更好的方式是检测 sort 是不是一个函数。（检测、排序）

```
//这样更好：检查 sort 是不是函数
function isSortable(object){
 return typeof object.sort == "function";
}
```

4. 在可能的情况下，要尽量使用 typeof 进行能力检测。（typeof、能力检测）
5. 检测某个或某几个特性并不能够确定浏览器。
6. 如果你知道自己的应用程序需要使用某些特定的浏览器特性，那么最好是一次性检测所有相关特性，而不要分别检测。看下面的例子。（一次性检测所有特性）

```
//确定浏览器是否支持 Netscape 风格的插件
```

```
var hasNSPlugins = !!(navigator.plugins && navigator.plugins.length);

//确定浏览器是否具有 DOM1 级规定的能力
var hasDOM1 = !!(document.getElementById && document.createElement &&
 document.getElementsByTagName);
```

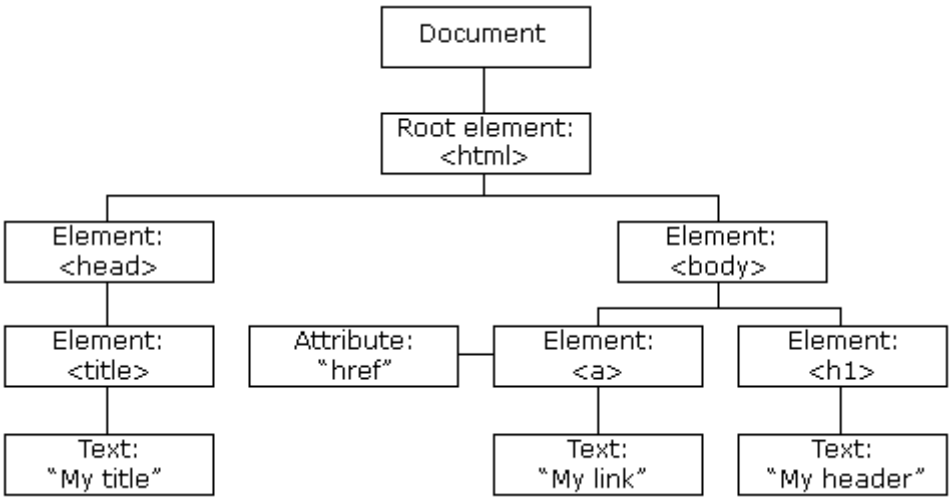
7. 怪癖检测的目标是识别浏览器的特殊行为。但与能力检测确认浏览器支持什么能力不同，怪癖检测是想要知道浏览器存在什么缺陷（“怪癖”也就是 bug）。（浏览器特殊行为检测）
8. 用户代理检测通过检测用户代理字符串来确定实际使用的浏览器。（用户代理检测、检测浏览器）
9. 用户代理检测是客户端检测的最后一个选择。只要可能，都应该优先采用能力检测和怪癖检测。

第 10 章 DOM

1. DOM 可以将任何 HTML 或 XML 文档描绘成一个由多层节点构成的结构。节点分为几种不同的类型，每种类型分别表示文档中不同的信息及（或）标记。每个节点都拥有各自的特点、数据和方法，另外也与其他节点存在某种关系。以下面的 HTML 为例：

```
<html>
 <head>
 <title>Sample Page</title>
 </head>
 <body>
 <p>Hello World!</p>
 </body>
</html>
```

将这个简单的 HTML 文档表示为一个层次结构，如图所示：



- 文档节点是每个文档的根节点。在这个例子中，文档节点只有一个子节点，即<html>元素，我们称之为文档元素。（文档节点、根节点）
- 文档元素是文档的最外层元素，文档中的其他所有元素都包含在文档元素中。每个文档只能有一个文档元素。在 HTML 页面中，文档元素始终都是<html>元素。在 XML 中，没有预定义的元素，因此任何元素都可能成为文档元素。（文档元素、最外层元素、即<html>元素）

2. W3C DOM 标准可以分为 DOM1、DOM2、DOM3 三个版本：

- DOM1：主要定义的是 HTML 和 XML 文档的底层结构。
- DOM2 级核心：在 1 级核心的基础上构建，为节点添加了更多方法和属性；
- DOM2 级视图：为文档定义了基于样式信息的不同视图；
- DOM2 级事件：定义了如何以编程方式来访问和改变 CSS 样式信息；
- DOM2 级遍历和范围：引入了遍历 DOM 文档和选择其特定部分的新接口。
- DOM2 级 HTML：在 1 级 HTML 基础上构建，添加了更多属性、方法和新接口。
- DOM3 级又增加了 XPath 模块和加载与保存模块。

3. DOM2 级和 3 级的目的在于扩展 DOM API，以满足操作 XML 的所有需求，同时提供更好的错误处理及特性检测能力。
4. 每一段标记都可以通过树中的一个节点来表示：HTML 元素通过元素节点表示，特性（attribute）通过特性节点表示，文档类型通过文档类型节点表示，而注释则通过注释节点表示。总共有 12 种节点类型，这里介绍 9 种。

- Node 类型
- Document 类型
- Element 类型
- Text 类型
- Comment 类型
- CDATASection 类型
- DocumentType 类型
- DocumentFragmen 类型
- Attr 类型

5. Node 对象是整个 DOM 的主要数据类型,代表文档树中的一个单独的节点。节点可以是元素节点、属性节点、文本节点等任何一种节点。
6. 每个节点都有一个 nodeType 属性，用于表明节点的类型。通过比较 nodeType 属性，可以很容易地确定节点的类型：（节点、nodeType 属性、节点类型）

```
if (someNode.nodeType == Node.ELEMENT_NODE){ //在 IE 中无效
 alert("Node is an element.");
}
```

然而，由于 IE 没有公开 Node 类型的构造函数，因此上面的代码在 IE 中会导致错误。为了确保跨浏览器兼容，最好还是将 nodeType 属性与数字值进行比较，如下所示：

```
if (someNode.nodeType == 1){ //适用于所有浏览器
 alert("Node is an element.");
}
```

7. nodeType：返回节点的类型。

   nodeName：返回节点的名称，根据其类型。

```
if (someNode.nodeType == 1){
 value = someNode.nodeName; //nodeName 的值是元素的标签名
}
```

8. childNodes：返回节点到子节点的节点列表，该列表是一个 NodeList 对象。NodeList 是一种类数组对象，用于保存一组有序的节点。
9. NodeList 中的节点：一上通过方括号，二是通过 item()方法：

```
var firstChild = someNode.childNodes[0];
var secondChild = someNode.childNodes.item(1);
var count = someNode.childNodes.length;
```

10. 每个节点都有一个 parentNode 属性，该属性指向文档树中的父节点。（节点 parentNode 属性、父节点）
11. 父节点的 firstChild 和 lastChild 属性分别指向其 childNodes 列表中的第一个和最后一个节点。（firstChild、第一个子节点、lastChild、最后一个子节点）
12. 节点方法：

(1) appendChild()：用于向 childNodes 列表的末尾添加一个节点。如果传入到 appendChild()中的节点已经是文档的一部分了，那结果就是将该节点从原来的位置转移到新位置。（appendChild()、添加节点）

```
var returnedNode = someNode.appendChild(newNode);
alert(returnedNode == newNode); //true
alert(someNode.lastChild == newNode); //true
```

(2) insertBefore()：将节点插入 childNodes 列表中某个特定的位置上。（insertBefore()、插入节点）

(3) replaceChild()：替换指定的节点。（replaceChild()、替换节点）

(4) removeChild()：移除指定的节点。（removeChild()、移除节点）

13. 有两个方法是所有类型的节点都有的：

(1) cloneNode()：用于创建调用这个方法的节点的一个完全相同的副本，也就是复制节点。（cloneNode()、复制节点）

(2) normalize()：处理文档树中的文本节点。由于解析器的实现或 DOM 操作等原因，可能会出现文本节点不包含文本，或者接连出现两个文本节点的情况。当在某个节点上调用这个方法时，就会在该节点的后代节点中查找上述两种情况。如果找到了空文本节点，则删除它；如果找到相邻的文本节点，则将它们合并为一个文本节点。（normalize()、处理文本节点）

14. Document 类型表示文档。在浏览器中，document 对象是 HTMLDocument（继承自 Document 类型）的一个实例，表示整个 HTML 页面。而且，document 对象是 window 对象的一个属性，因此可以将其作为全局对象来访问。（Document 类型表示文档）

15. Document 节点有两个内置的访问其子节点的快捷方式：

- documentElement 属性，该属性始终指向 HTML 页面中的<html>元素。
- childNodes 列表访问文档元素。通过 documentElement 属性则能更快捷、更直接地访问该元素。

16. document 对象还有一个 body 属性，直接指向<body>元素。（document 对象、body 属性、<body>元素）

```
var body = document.body; //取得对<body>的引用
```

17. Document 另一个可能的子节点是 DocumentType。通常将<!DOCTYPE>标签看成一个与文档其他部分不同的实体，可以通过 doctype 属性（在浏览器中是 document.doctype）来访问它的信息。

18. 从技术上说，出现在<html>元素外部的注释应该算是文档的子节点。然而，不同的浏览器在是否解析这些注释以及能否正确处理它们等方面，也存在很大差异。

```
<!--第一条注释 -->
<html>
 <body>

 </body>
</html>
<!--第二条注释 -->
```

19. 作为 HTMLDocument 的一个实例，document 对象还有一些标准的 Document 对象所没有的属性。

- title 属性：包含着<title>元素中的文本。
- URL 属性：包含页面完整的 URL（即地址栏中显示的 URL）。
- domain 属性：只包含页面的域名。该属性可以设置。但由于安全方面的限制，也并非可以给 domain 设置任何值。如果 URL 中包含一个子域名，例如 p2p.wrox.com，那么就只能将 domain 设置为"wrox.com"。不能将这个属性设置为 URL 中不包含的域。

➢ 这里要注意，domain 中不同的二级域名属于不同的域。（二级域名、不同的域）

- referrer 属性：保存着链接到当前页面的那个页面的 URL。

20. 由于跨域安全限制，来自不同子域的页面无法通过 JavaScript 通信。而通过将每个页面的 document.domain 设置为相同的值，这些页面就可以互相访问对方包含的 JavaScript 对象了。（跨域安全限制、domain 设置为相同值、互相访问）

例如，假设有一个页面加载自 www.wrox.com，其中包含一个内嵌框架，框架内的页面加载自 p2p.wrox.com。由于 document.domain 字符串不一样，内外两个页面之间无法相互访问对方的 JavaScript 对象。但如果将这两个页面的 document.domain 值都设置为"wrox.com"，它们之间就可以通信了。

21. 浏览器对 domain 属性还有一个限制，即如果域名一开始是“松散的”（loose），那么不能将它再设置为“紧绷的”（tight）。换句话说，在将 document.domain 设置为"wrox.com"之后，就不能再将其设置回"p2p.wrox.com"，否则将会导致错误，如下面的例子所示。（不能再设置回）

```
//假设页面来自于 p2p.wrox.com 域

document.domain = "wrox.com"; //松散的（成功）

document.domain = "p2p.wrox.com"; //紧绷的（出错！）
```

22. Document 类型提供了几个方法用于查找元素：

- getElementById() 方法可返回对拥有指定 ID 的第一个对象的引用。
- getElementsByTagName() 方法可返回带有指定标签名的对象的集合。
- getElementsByName() 方法可返回带有指定名称的对象的集合。

23. document 对象还有一些特殊的集合：

- document.anchors，包含文档中所有带 name 特性的<a>元素；
- document.forms，包含文档中所有的<form>元素，与 document.getElementsByTagName("form")得到的结果相同；
- document.images，包含文档中所有的<img>元素，与 document.getElementsByTagName("img")得到的结果相同；
- document.links，包含文档中所有带 href 特性的<a>元素。

24. document.implementation 属性可返回处理该文档的 DOMImplementation 对象。该对象可以用于检测浏览器实现了 DOM 的哪些部分。

25. DOM1 级只为 document.implementation 规定了一个方法，即 hasFeature()，该函数用于检测浏览器是否支持给定名称和版本的功能。（hasFeature、浏览器是否支持特定功能）

```
var hasXmlDom = document.implementation.hasFeature("XML", "1.0");
```

26. 但是，hasFeature()返回 true 有时候也不意味着实现与规范一致。为此，我们建议多数情况下，在使用 DOM 的某些特殊的功能之前，最好除了检测 hasFeature()之外，还同

时使用能力检测。（hasFeature()检测加能力检测）

27. 其余的 Document 方法：

- open()：打开一个流，以收集来自任何 document.write() 或 document.writeln() 方法的输出。（open()、打开流）
- close()：关闭用 document.open() 方法打开的输出流，并显示选定的数据。（close()、关闭输出流）
- write()：向文档写 HTML 表达式 或 JavaScript 代码。（write()、写）

```
<html>
<head>
 <title>document.write() Example</title>
</head>
<body>
 <p>The current date and time is:
 <script type="text/javascript">
 document.write("<strong>" + (new Date()).toString() + "</strong>");
 </script>
 </p>
</body>
</html>
```

- writeln()：等同于 write() 方法，不同的是在每个表达式之后写一个换行符。（writeln()、写、加换行符）

28. 使用 write()和 writeln()方法动态地包含外部资源，例如 JavaScript 文件等。在包含 JavaScript 文件时，必须注意不能直接包含字符串"</script>"，因为这会导致该字符串被解释为脚本块的结束，它后面的代码将无法执行。（write()、writeln()、不能直接包含</script>）

```
<html>
<head>
 <title>document.write() Example 3</title>
</head>
<body>
 <script type="text/javascript">
 document.write("<script type=\"text/javascript\" src=\"file.js\">" +
 "<\/script>");
 </script>
</body>
</html>
```

29. HTML DOM 中，Element 对象表示 HTML 元素。

30. element.nodeName 属性：返回元素的名称。

element.tagName 属性：返回元素的标签名。这两个属性会返回相同的值。以下列元素为例：

```
<div id="myDiv"></div>
```

可以像下面这样取得这个元素及其标签名：

```
var div = document.getElementById("myDiv");
alert(div.tagName); //"DIV"
alert(div.tagName == div.nodeName); //true
```

在 HTML 中，标签名始终都以全部大写表示，所以 tagName 返回的是大写。

31. element.getAttribute()：返回元素节点的指定属性值。

element.setAttribute()：把指定属性设置或更改为指定值。添加一个自定义的属性：

```
div.mycolor = "red";
alert(div.getAttribute("mycolor")); //null（IE 除外）
```

element.removeAttribute()：从元素中移除指定属性。

```
var div = document.getElementById("myDiv");
alert(div.getAttribute("id")); //"myDiv"
alert(div.getAttribute("class")); //"bd"
alert(div.getAttribute("title")); //"Body text"
alert(div.getAttribute("lang")); //"en"
alert(div.getAttribute("dir")); //"ltr"
```

32. 有两类特殊的特性，它们虽然有对应的属性名，但属性的值与通过 getAttribute()返回的值并不相同。

- style：style 特性值中包含的是 CSS 文本，而通过属性来访问它则会返回一个对象。
- onclick：onclick 特性中包含的是 JavaScript 代码，如果通过 getAttribute()访问，则会返回相应代码的字符串。而在访问 onclick 属性时，则会返回一个 JavaScript。函数

由于存在这些差别，在通过 JavaScript 以编程方式操作 DOM 时，开发人员经常不使用 getAttribute()，而是只使用对象的属性。只有在取得自定义特性值的情况下，才会使用 getAttribute()方法。

33. element.attributes 属性返回指定节点的属性集合，即 NamedNodeMap。DOM 节点类型中只有 Element 类型使用 attributes 属性。

34. NamedNodeMap 对象拥有下列方法。

- getNamedItem_(name)_：返回指定 nodeName 属性的节点；
- removeNamedItem_(name)_：从列表中移除指定 nodeName 属性的节点；
- setNamedItem_(node)_：向列表中添加节点，以节点的 nodeName 属性为索引；
- item_(pos)_：返回位于数字 pos 位置处的节点。

35. element.attributes 属性中包含一系列节点，每个节点的 nodeName 就是属性的名称，而

节点的 nodeValue 就是属性的值。要取得元素的 id 属性，可以使用以下代码。

```
var id = element.attributes.getNamedItem("id").nodeValue;
```

36. document.createElement()：创建新元素。

```
var div = document.createElement("div");
div.id = "myNewDiv";
div.className = "box";
```

37. text 属性可设置或返回选项的文本值。

38. 可以操作节点中的文本。

- appendData(text)：将 text 添加到节点的末尾。
- deleteData(offset, count)：从 offset 指定的位置开始删除 count 个字符。
- insertData(offset, text)：在 offset 指定的位置插入 text。
- replaceData(offset, count, text)：用 text 替换从 offset 指定的位置开始到 offset+ count 为止处的文本。
- splitText(offset)：从 offset 指定的位置将当前文本节点分割成两个文本节点。
- substringData(offset, count)：提取从 offset 指定的位置开始到 offset+count 为止处的字符串。

39. 文本节点有一个 length 属性，保存着节点中字符的数目。而且，nodeValue.length 和 data.length 中也保存着同样的值。

40. document.createTextNode()：创建新文本节点。

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

document.body.appendChild(element);
```

41. element.normalize()：移除空的文本节点，并连接相邻的文本节点。

```
var element = document.createElement("div");
element.className = "message";

var textNode = document.createTextNode("Hello world!");
element.appendChild(textNode);

var anotherTextNode = document.createTextNode("Yippee!");
element.appendChild(anotherTextNode);
```

```
document.body.appendChild(element);

alert(element.childNodes.length); //2

element.normalize();
alert(element.childNodes.length); //1
alert(element.firstChild.nodeValue); // "Hello world!Yippee!"
```

此处将两段文本合并成一段文本。

42. splitText()：按照指定的 offset 把文本节点分割为两个节点。
43. Comment 对象表示文档中注释节点的内容。
44. Comment 类型与 Text 类型继承自相同的基类，因此它拥有除 splitText()之外的所有字符串操作方法。

```
<div id="myDiv"><!--A comment --></div>
```

在此，注释节点是 元素的一个子节点，因此可以通过下面的代码来访问它。

```
var div = document.getElementById("myDiv");
var comment = div.firstChild;
alert(comment.data); //"A comment"
```

45. Document.createComment()：创建一个注释对象。
46. 与 Comment 类似，CDATASection 类型继承自 Text 类型，因此拥有除 splitText()之外的所有字符串操作方法。
47. DocumentTyp 类型包含着与文档的 doctype 有关的所有信息。
48. DocumentFragment 表示一个没有父级文件的最小文档对象。它被当做一个轻量版的 Document 使用，用于存储已排好版的或尚未打理好格式的 XML 片段。最大的区别是因为 DocumentFragment 不是真实 DOM 树的一部分，它的变化不会引起 DOM 树的重新渲染的操作(reflow)，且不会导致性能等问题。
49. DocumentFragment 继承了 Node 的所有方法，通常用于执行那些针对文档的 DOM 操作。如果将文档中的节点添加到 DocumentFragment 中，就会从文档树中移除该节点，也不会从浏览器中再看到该节点。添加到 DocumentFragment 中的新节点同样也不属于文档树。

```
var fragment = document.createDocumentFragment();
var ul = document.getElementById("myList");
var li = null;

for (var i=0; i < 3; i++){
 li = document.createElement("li");
 li.appendChild(document.createTextNode("Item " + (i+1)));
 fragment.appendChild(li);
}
```

```
ul.appendChild(fragment);
```

50. Attr 对象（特性）表示 Element 对象的属性。

51. Attr 对象有 3 个属性：name、value 和 specified。其中，name 是特性名称（与 nodeName 的值相同），value 是特性的值（与 nodeValue 的值相同），而 specified 是一个布尔值，用以区别特性是在代码中指定的，还是默认的。（

52. 使用 document.createAttribute()并传入特性的名称可以创建新的特性节点。例如，要为元素添加 align 特性，可以使用下列代码：

```
var attr = document.createAttribute("align");
attr.value = "left";
element.setAttributeNode(attr);
alert(element.attributes["align"].value); //"left"
alert(element.getAttributeNode("align").value); //"left"
alert(element.getAttribute("align")); //"left"
```

53. 动态脚本，指的是在页面加载时不存在，但将来的某一时刻通过修改 DOM 动态添加的脚本。

54. 跟操作 HTML 元素一样，创建动态脚本也有两种方式：插入外部文件和直接插入 JavaScript 代码。

(1) 动态加载的外部 JavaScript 文件能够立即运行，比如下面的<script>元素：

```
<script type="text/javascript" src="client.js"></script>
```

(2) 也可以把这个元素添加到<head>元素中，效果相同。

55. 能够把 CSS 样式包含到 HTML 页面中的元素有两个。其中，<link>元素用于包含来自外部的文件，而<style>元素用于指定嵌入的样式。

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

使用 DOM 代码可以很容易地动态创建出这个元素：

```
var link = document.createElement("link");
link.rel = "stylesheet";
link.type = "text/css";
link.href = "style.css";
var head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

56. 加载外部样式文件的过程是异步的，也就是加载样式与执行 JavaScript 代码的过程没有固定的次序。（加载外部文件、异步、没有固定次序）

57. 使用 NodeList 对象时要注意，下列代码会导致无限循环：

```
var divs = document.getElementsByTagName("div"),
for (i=0; i < divs.length; i++){
```

```
 div = document.createElement("div");
 document.body.appendChild(div);
}
```

关键在于，每次循环，都需要算出 div.length 的值，而在循环中又会不断更新该值。

第 11 章 DOM 扩展

1. 对 DOM 的两个主要的扩展是 Selectors API（选择符 API）和 HTML5。
2. querySelector()查找指定的元素。

```
//取得 body 元素
var body = document.querySelector("body");

//取得 ID 为"myDiv"的元素
var myDiv = document.querySelector("#myDiv");

//取得类为"selected"的第一个元素
var selected = document.querySelector(".selected");

//取得类为"button"的第一个图像元素
var img = document.body.querySelector("img.button");
```

3. querySelectorAll()方法返回的是所有匹配的元素而不仅仅是一个元素。这个方法返回的是一个 NodeList 的实例。
4. matchesSelector()：接收一个参数，即 CSS 选择符，如果调用元素与该选择符匹配，返回 true；否则，返回 false。（matchesSelector()、调用元素与该选择符匹配）

```
if (document.body.matchesSelector("body.page1")){
 //true
}
```

5. Element Tranversal API 为 DOM 元素添加了以下 5 个属性

- childElementCount：返回子元素(不包括文本节点和注释)的个数。
- firstElementChild：指向第一个子元素；firstChild 的元素版。
- lastElementChild：指向最后一个子元素；lastChild 的元素版。
- previousElementSibling：指向前一个同辈元素；previousSibling 的元素版。
- nexrtElementSibling：指向后一个同辈元素；nextSibling 的元素版。

支持的浏览器为 DOM 元素添加了这些属性，利用这些元素不必担心空白文本节点，从而可以更方便地查找 DOM 元素了。

```
var i,len,child = element.firstChild;
while(child != element.lastChild){
```

```
 if (child.nodeType == 1){ //检查是不是元素
 processChild(child);
 }
 child = child.nextSibling;
}
```

而使用 Element Traversal 新增的元素，代码会更简洁。

```
var i,len,child = element.firstElementChild;
while(child != element.lastElementChild){
 processChild(child); //已知其是元素
 child = child.nextElementSibling;
}
```

6. getElementsByClassName()：查找带有相同类名的所有 HTML 元素。

```
var allCurrentUsernames = document.getElementsByClassName("username current");

//取得 ID 为"myDiv"的元素中带有类名"selected"的所有元素
var selected =
document.getElementById("myDiv").getElementsByClassName("selected");
```

7. classList 属性返回元素类名的列表，该列表是一个 DOMTokenList 对象。DOMTokenList 有一个表示自己包含多少元素的 length 属性，而要取得每个元素可以使用 item()方法，也可以使用方括号语法。

- add(value)：将给定的字符串值添加到列表中。如果值已经存在，就不添加了。
- contains(value)：表示列表中是否存在给定的值，如果存在则返回 true，否则返回 false。
- remove(value)：从列表中删除给定的字符串。
- toggle(value)：如果列表中已经存在给定的值，删除它；如果列表中没有给定的值，添加它。

```
<div class="bd user disabled">...</div>
```

这里有 3 个类名，要删除 user 类名，只需要一行代码：

```
div.classList.remove("user");
```

8. document.activeElement 属性返回文档中当前获得焦点的元素。
9. focus()方法：获取焦点。
10. 默认情况下，文档刚刚加载完成时，document.activeElement 中保存的是 document.body 元素的引用。文档加载期间，document.activeElement 的值为 null。
11. 新增的 document.hasFocus()方法用于确定文档是否获得了焦点。

```
var button = document.getElementById("myButton");
button.focus();
```

```
alert(document.hasFocus()); //true
```

12. HTMLDocument 扩展：

(1) readyState 属性有两个可能的值：

- loading，正在加载文档；
- complete，已经加载完文档。

```
if (document.readyState == "complete"){
 //执行操作
}
```

(2) document 添加了一个名为 compatMode 的属性，该属性保存了浏览器采用了的渲染模式。在标准模式下，document.compatMode 的值等于"CSS1Compat"，而在混杂模式下，document.compatMode 的值等于"BackCompat"。（compatMode、浏览器采用了的渲染模式）

```
if (document.compatMode == "CSS1Compat"){
 alert("Standards mode");
} else {
 alert("Quirks mode");
}
```

(3) HTML5 新增了 document.head 属性，引用文档的<head>元素。

```
var head = document.head || document.getElementsByTagName("head")[0];
```

(4) HTML5 新增了几个与文档字符集有关的属性。其中，charset 属性表示文档中实际使用的字符集，也可以用来指定新字符集。默认情况下，这个属性的值为"UTF-16"，但可以通过<meta>元素、响应头部或直接设置 charset 属性修改这个值。

```
alert(document.charset); //"UTF-16"
document.charset = "UTF-8";
```

另一个属性是 defaultCharset，保存当前文档默认的字符集。（defaultCharset、默认字符集）

(5) HTML5 规定可以为元素添加非标准的属性，但要添加前缀 data-，目的是为元素提供与渲染无关的信息，或者提供语义信息。这些属性可以任意添加、随便命名，只要以 data-开头即可。（非标准属性、前缀 data-）

```
<div id="myDiv" data-appId="12345" data-myname="Nicholas"></div>
```

添加了自定义属性之后，可以通过元素的 dataset 属性来访问自定义属性的值。dataset 属性的值是 DOMStringMap 的一个实例，也就是一个名值对儿的映射。

```
//本例中使用的方法仅用于演示

var div = document.getElementById("myDiv");
```

```
//取得自定义属性的值
var appId = div.dataset.appId;
var myName = div.dataset.myname;

//设置值
div.dataset.appId = 23456;
div.dataset.myname = "Michael";

//有没有"myname"值呢?
if (div.dataset.myname){
 alert("Hello, " + div.dataset.myname);
}
```

(6) innerHTML 属性设置或返回表格行的开始和结束标签之间的 HTML。

```
<html>
<head>
<script type="text/javascript">
function getInnerHTML()
  {
  alert(document.getElementById("tr1").innerHTML);
  }
</script>
</head>
<body>

<table border="1">
<tr id="tr1">
<th>Firstname</th>
<th>Lastname</th>
</tr>
<tr id="tr2">
<td>Peter</td>
<td>Griffin</td>
</tr>
</table>
<br />
<input type="button" onclick="getInnerHTML()"
value="Alert innerHTML of table row" />

</body>
</html>
```

输出结果如下：


```
<html>
 <head>
 <script>
 function changeLink()
 {
 document.getElementById('myAnchor').innerHTML="W3Schools";
 document.getElementById('myAnchor').href="http://www.w3schools.com";
 document.getElementById('myAnchor').target="_blank";
 }
 </script>
 </head>
 <body>

 <a id="myAnchor" href="http://www.microsoft.com">Microsoft</a>
 <input type="button" onclick="changeLink()" value="Change link">

 </body>
</html>
```

该例子用于改变链接的文本、URL 和 target。

(7) outerHTML 属性：在读模式下，outerHTML 返回调用它的元素及所有子节点的 HTML 标签。在写模式下，outerHTML 会根据指定的 HTML 字符串创建新的 DOM 子树，然后用这个 DOM 子树完全替换调用元素。

(8) scrollIntoView()可以在所有 HTML 元素上调用，通过滚动浏览器窗口或某个容器元素，调用元素就可以出现在视口中。（scrollIntoView()、滚动窗口）

第 12 章 DOM2 和 DOM3

1. “DOM2 级核心”没有引入新类型，它只是在 DOM1 级的基础上通过增加新方法和

新属性来增强了既有类型。“DOM3 级核心”同样增强了既有类型，但也引入了一些新类型。

2. HTML 不支持 XML 命名空间，但 XHTML 支持 XML 命名空间。
3. 我们可以通过下列代码来确定浏览器是否支持这些 DOM 模块：

```
var supportsDOM2Core = document.implementation.hasFeature("Core","2.0");
var supportsDOM3Core = document.implementation.hasFeature("Core","3.0");
var supportsDOM2HTML = document.implementation.hasFeature("HTML","2.0");
var supportsDOM2Views = document.implementation.hasFeature("Views","2.0");
var supportsDOM2XML = document.implementation.hasFeature("XML","2.0");
```

4. 现代的 JS 写法建议你始终使用前置 var 声明所有变量。
5. 在 HTML 中定义样式的方式有 3 种：通过<link/>元素包含外部样式表文件、使用<style/>元素定义嵌入式样式，以及使用 style 特性定义针对特定元素的样式。
6. “DOM2 级样式”模块围绕这 3 种应用样式的机制提供了一套 API。要确定浏览器是否支持 DOM2 级定义的 CSS 能力，可以使用下列代码。

```
var supportsDOM2CSS = document.implementation.hasFeature("CSS", "2.0");
var supportsDOM2CSS2 = document.implementation.hasFeature("CSS2", "2.0");
```

7. 当 HTML 元素存在 style 属性时，该元素在 JavaScript 中都有一个对应的 style 属性。（例如：background-image 对应 style.backgroundImage）

注：CSS 属性中的 float 属性，是 JavaScript 中的保留字，因此不能用作属性名，DOM2 中使用的是 cssFloat 属性名。

```
var myDiv = document.getElementById("myDiv");

//设置背景颜色
myDiv.style.backgroundColor = "red";

//改变大小
myDiv.style.width = "100px";
myDiv.style.height = "200px";

//指定边框
myDiv.style.border = "1px solid black";
```

此例也可以使用 setProperty()来实现：

```
  var myDiv = document.getElementByIdx_x("myDiv");

            //set the background color
            myDiv.style.setProperty("background-color", "red", "");;

            //change the dimensions
```

```
            myDiv.style.setProperty("width", "100px", "");
            myDiv.style.setProperty("height", "200px", "");

            //assign a border
            myDiv.style.setProperty("border", "1px solid black", "");
```

8. “DOM2 级样式”规范还为 style 对象定义了一些属性和方法。这些属性和方法在提供元素的 style 特性值的同时，也可以修改样式。下面列出了这些属性和方法。

(1) cssText：如前所述，通过它能够访问到 style 特性中的 CSS 代码。

(2) length：应用给元素的 CSS 属性的数量。

(3) parentRule：表示 CSS 信息的 CSSRule 对象。本节后面将讨论 CSSRule 类型。

(4) getPropertyCSSValue(propertyName)：返回包含给定属性值的 CSSValue 对象。

(5) getPropertyPriority(propertyName)：如果给定的属性使用了!important 设置，则返回"important"；否则，返回空字符串。

(6) getPropertyValue(propertyName)：返回给定属性的字符串值。

(7) item(index)：返回给定位置的 CSS 属性的名称。

(8) removeProperty(propertyName)：从样式中删除给定属性。

(9) setProperty(propertyName,value,priority)：将给定属性设置为相应的值，并加上优先权标志（"important"或者一个空字符串）。

9. getComputedStyle()：获取 css 属性值。

10. CSSStyleSheet 类型表示的是样式表，包括通过<link>元素包含的样式表和在<style>元素中定义的样式表。使用下面的代码可以确定浏览器是否支持 DOM2 级样式表。

```
var supportsDOM2StyleSheets =document.implementation.hasFeature("StyleSheets",
"2.0");
```

11. CSSRule 对象表示样式表中的每一条规则。CSSStyleRule 对象包含下列属性。

- cssText：返回整条规则对应的文本。由于浏览器对样式表的内部处理方式不同，返回的文本可能会与样式表中实际的文本不一样；Safari 始终都会将文本转换成全部小写。IE 不支持这个属性。
- parentRule：如果当前规则是导入的规则，这个属性引用的就是导入规则；否则，这个值为 null。IE 不支持这个属性。
- parentStyleSheet：当前规则所属的样式表。IE 不支持这个属性。
- selectorText：返回当前规则的选择符文本。由于浏览器对样式表的内部处理方式不同，返回的文本可能会与样式表中实际的文本不一样（例如，Safari 3 之前的版本始终会将文本转换成全部小写）。在 Firefox、Safari、Chrome 和 IE 中这个属性是只读的。Opera 允许修改 selectorText。
- style：一个 CSSStyleDeclaration 对象，可以通过它设置和取得规则中特定的样式值。

- type：表示规则类型的常量值。对于样式规则，这个值是 1。IE 不支持这个属性。

12. DOM 规定，要向现有样式表中添加新规则，需要使用 insertRule()方法。

```
sheet.insertRule("body { background-color: silver }", 0); //DOM 方法
```

13. 从样式表中删除规则的方法是 deleteRule()，这个方法接受一个参数：要删除的规则的位置。例如，要删除样式表中的第一条规则，可以使用以下代码。

```
sheet.deleteRule(0); //DOM 方法
```

14. 通过下列 4 个属性可以取得元素的偏移量。

- offsetHeight：元素在垂直方向上占用的空间大小，单位像素。包括元素的高度、（可见的）水平滚动条的高度、上边框高度和下边框高度。
- offsetWidth：元素在水平方向上占用的空间大小，单位像素。包括元素的宽度、（可见的）垂直滚动条的宽度、左边框宽度和右边框宽度。
- offsetLeft：元素的左外边框至包含元素的左内边框之间的像素距离。
- offsetTop：元素的上外边框至包含元素的上内边框之间的像素距离。

```
function getElementTop(element) {
      var actualTop = element.offsetTop;
      var current = element.offsetParent;
      while (current !== null) {
            actualTop += current.offsetTop;
            current = current.offsetParent;
      }
      return actualTop;
}
```

15. 元素的客户区大小（client dimension），指的是元素内容及其内边距所占据的空间大小。有关客户区大小的属性有两个：clientWidth 和 clientHeight。其中，clientWidth 属性是元素内容区宽度加上左右内边距宽度；clientHeight 属性是元素内容区高度加上上下内边距高度。

```
function getViewport() {
      if (document.compatMode == "BackCompat") {
            return {
                  width: document.body.clientWidth,
                  height: document.body.clientHeight
            };
      } else {
            return {
                  width: document.documentElement.clientWidth,
                  height: document.documentElement.clientHeight
            };
```

```
    }
}
```

16. 滚动大小相关的属性。

- scrollHeight：在没有滚动条的情况下，元素内容的总高度。
- scrollWidth：在没有滚动条的情况下，元素内容的总宽度。
- scrollLeft：被隐藏在内容区域左侧的像素数。通过设置这个属性可以改变元素的滚动位置。
- scrollTop：被隐藏在内容区域上方的像素数。通过设置这个属性可以改变元素的滚动位置。

17. “DOM2 级遍历和范围”模块定义了两个用于顺序遍历 DOM 结构的类型：NodeIterator 和 TreeWalker。这两个类型能够基于给定的起点对 DOM 结构执行深度优先（depth-first）的遍历操作。

18. 使用下列代码可以检测浏览器对 DOM2 级遍历能力的支持情况。

```
var supportsTraversals = document.implementation.hasFeature("Traversal", "2.0");
var supportsNodeIterator = (typeof document.createNodeIterator == "function");
var supportsTreeWalker = (typeof document.createTreeWalker == "function");
```

19. NodeIterator 类型是两者中比较简单的一个，可以使用 document.createNodeIterator()方法创建它的新实例。

```
var div = document.getElementById("div1");
var filter = function(node){
 return node.tagName.toLowerCase() == "li" ?
 NodeFilter.FILTER_ACCEPT :
 NodeFilter.FILTER_SKIP;
};

var iterator = document.createNodeIterator(div, NodeFilter.SHOW_ELEMENT,
 filter, false);

var node = iterator.nextNode();
while (node !== null) {
 alert(node.tagName); //输出标签名
 node = iterator.nextNode();
}
```

返回遍历中遇到的<li>元素。

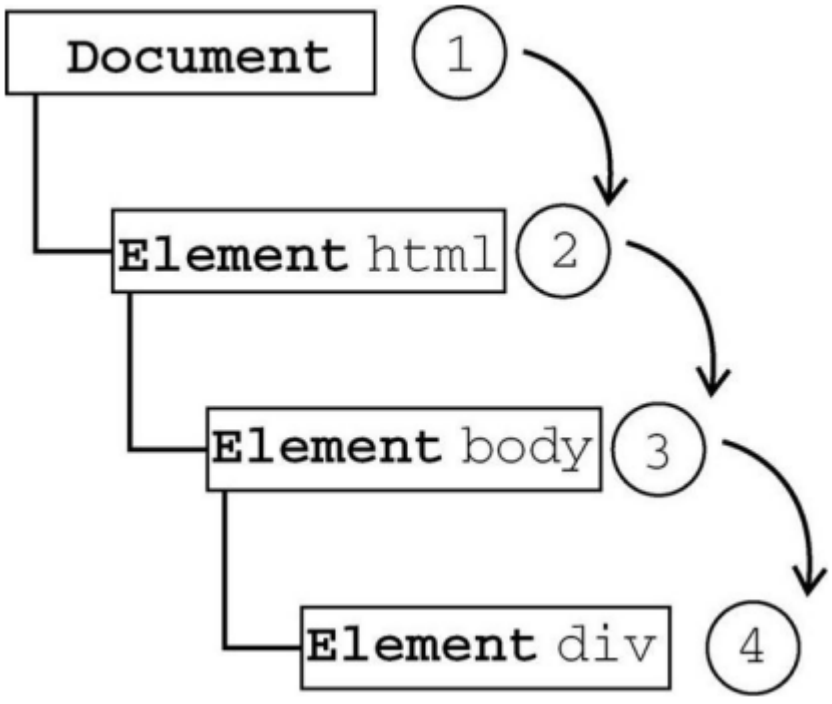
20. TreeWalker 是 NodeIterator 的一个更高级的版本。除了包括 nextNode()和 previousNode()在内的相同的功能之外，这个类型还提供了下列用于在不同方向上遍历 DOM 结构的方法。

- parentNode()：遍历到当前节点的父节点；

- firstChild()：遍历到当前节点的第一个子节点；
- lastChild()：遍历到当前节点的最后一个子节点；
- nextSibling()：遍历到当前节点的下一个同辈节点；
- previousSibling()：遍历到当前节点的上一个同辈节点。

第 13 章 事件

1. JavaScript 与 HTML 之间的交互是通过事件实现的。
2. 如果你单击了某个按钮，他们都认为单击事件不仅仅发生在按钮上。换句话说，在单击按钮的同时，你也单击了按钮的容器元素，甚至也单击了整个页面。
3. 事件流描述的是从页面中接收事件的顺序。
4. IE 的事件流叫做事件冒泡，即事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点（文档）。
5. Netscape Communicator 团队提出的另一种事件流叫做事件捕获（event capturing）。事件捕获的思想是不太具体的节点应该更早接收到事件，而最具体的节点应该最后接收到事件。事件捕获的用意在于在事件到达预定目标之前捕获它。



6. 事件处理程序的名字以"on"开头，因此 click 事件的事件处理程序就是 onclick，load 事件的事件处理程序就是 onload。
7. 要在按钮被单击时执行一些 JavaScript，可以像下面这样编写代码：

```
<input type="button" value="Click Me" onclick="alert('Clicked')" />
```

当单击这个按钮时，就会显示一个警告框，警告框文本为“Clicked”。

不能在其中使用未经转义的 HTML 语法字符，例如和号（&）、双引号（""）、小于

号（<）或大于号（>）。为了避免使用 HTML 实体，这里使用了单引号。如果想要使用双引号，那么就要将代码改写成如下所示：

```
<input type="button" value="Click Me" onclick="alert(&quot;Clicked&quot;)" />
```

8. 在 HTML 中定义的事件处理程序可以包含要执行的具体动作，也可以调用在页面其他地方定义的脚本，如下面的例子所示：

```
<script type="text/javascript">
function showMessage(){
    alert("Hello world!");
}
</script>
<input type="button" value="Click Me" onclick="showMessage()" />
```

9. 事件处理程序具有一些独到之处

(1) 变量 event，也就是事件对象：（event、变量对象）

```
<!-- 输出 "click" -->
<input type="button" value="Click Me" onclick="alert(event.type)">
```

通过 event 变量，可以直接访问事件对象，你不用自己定义它，也不用从函数的参数列表中读取。

(2) this 值等于事件的目标元素：（this、事件的目标元素）

```
<!-- 输出 "Click Me" -->
<input type="button" value="Click Me" onclick="alert(this.value)">
```

这个 this 值也就是 input 元素。

10. 每个元素（包括 window 和 document）都有自己的事件处理程序属性，这些属性通常全部小写，例如 onclick。将这种属性的值设置为一个函数，就可以指定事件处理程序，如下所示：

```
var btn = document.getElementById("myBtn");
btn.onclick = function(){
    alert("Clicked");
};
```

11. addEventListener()添加事件处理程序。

removeEventListener()：删除事件处理程序。

要在按钮上为 click 事件添加事件处理程序，可以使用下列代码：

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click", function(){
    alert(this.id);
}, false);
```

如果为元素添加多个信号处理程序，则信号处理程序会按顺序被触发：（多个信号处

理程序、顺序触发）

```
var btn = document.getElementById("myBtn");
btn.addEventListener("click",
function() {
      alert(this.id);
},
false);
btn.addEventListener("click",
function() {
      alert("Hello world!");
},
f alse);
```

12. event 对象的 target 属性指的是事件目标。在事件处理程序内部，对象 this 始终等于 currentTarget 的值，而 target 则只包含事件的实际目标。如果直接将事件处理程序指定给了目标元素，则 this、currentTarget 和 target 包含相同的值。

```
var btn = document.getElementById("myBtn");
btn.onclick = function(event) {
      alert(event.currentTarget === this); //true
      alert(event.target === this); //true
```

13. UI 事件指的是那些不一定与用户操作有关的事件。现有的 UI 事件如下。

(1) DOMActivate：表示元素已经被用户操作（通过鼠标或键盘）激活。这个事件在 DOM3 级事件中被废弃，但 Firefox 2+和 Chrome 支持它。考虑到不同浏览器实现的差异，不建议使用这个事件。（DOMActivate、已经被激活）

(2) load：当页面完全加载后在 window 上面触发，当所有框架都加载完毕时在框架集上面触发，当图像加载完毕时在<img>元素上面触发，或者当嵌入的内容加载完毕时在<object>元素上面触发。（load、页面完全加载）

(3) unload：当页面完全卸载后在 window 上面触发，当所有框架都卸载后在框架集上面触发，或者当嵌入的内容卸载完毕后在<object>元素上面触发。（unload、页面完全卸载）

(4) abort：在用户停止下载过程时，如果嵌入的内容没有加载完，则在<object>元素上面触发。（abort、没有加载完便停）

(5) error：当发生 JavaScript 错误时在 window 上面触发，当无法加载图像时在<img>元素上面触发，当无法加载嵌入内容时在<object>元素上面触发，或者当有一或多个框架无法加载时在框架集上面触发。第 17 章将继续讨论这个事件。（error、发生错误）

(6) select：当用户选择文本框（<input>或<texterea>）中的一或多个字符时触发。第 14 章将继续讨论这个事件。（select、当用户选择文本时）

(7) resize：当窗口或框架的大小变化时在 window 或框架上面触发。（resize、窗口大小变化时）

(8) scroll：当用户滚动带滚动条的元素中的内容时，在该元素上面触发。<body>元素中包含所加载页面的滚动条。（scroll、滚动时）

除了 DOMActivate 之外，其他事件在 DOM2 级事件中都归为 HTML 事件（DOMActivate 在 DOM2 级中仍然属于 UI 事件）。要确定浏览器是否支持 DOM2 级事件规定的 HTML 事件，可以使用如下代码：

```
var isSupported = document.implementation.hasFeature("HTMLEvents", "2.0");
```

注意，只有根据“DOM2 级事件”实现这些事件的浏览器才会返回 true。而以非标准方式支持这些事件的浏览器则会返回 false。

14. 当浏览器窗口被调整到一个新的高度或宽度时，就会触发 resize 事件。

15. 焦点事件会在页面元素获得或失去焦点时触发。利用这些事件并与 document.hasFocus()方法及 document.activeElement 属性配合，可以知晓用户在页面上的行踪。有以下 6 个焦点事件。

- blur：在元素失去焦点时触发。这个事件不会冒泡；所有浏览器都支持它。（blur、失去焦点、不会冒泡）
- DOMFocusIn：在元素获得焦点时触发。这个事件与 HTML 事件 focus 等价，但它冒泡。只有 Opera 支持这个事件。DOM3 级事件废弃了 DOMFocusIn，选择了 focusin。（DOMFocusIn、获得焦点时）
- focus：在元素获得焦点时触发。这个事件不会冒泡；所有浏览器都支持它。（focus、获得焦点时、不会冒泡）
- focusin：在元素获得焦点时触发。这个事件与 HTML 事件 focus 等价，但它冒泡。支持这个事件的浏览器有 IE5.5+、Safari 5.1+、Opera 11.5+和 Chrome。（focusin、获得焦点）
- focusout：在元素失去焦点时触发。这个事件是 HTML 事件 blur 的通用版本。支持这个事件的浏览器有 IE5.5+、Safari 5.1+、Opera 11.5+和 Chrome。（focusout、失去焦点）

16. DOM3 级事件中定义了 9 个鼠标事件，简介如下。（鼠标事件）

- click：在用户单击主鼠标按钮（一般是左边的按钮）或者按下回车键时触发。这一点对确保易访问性很重要，意味着 onclick 事件处理程序既可以通过键盘也可以通过鼠标执行。
- dblclick：在用户双击主鼠标按钮（一般是左边的按钮）时触发。从技术上说，这个事件并不是 DOM2 级事件规范中规定的，但鉴于它得到了广泛支持，所以 DOM3 级事件将其纳入了标准。
- mousedown：在用户按下了任意鼠标按钮时触发。不能通过键盘触发这个事件。
- mouseenter：在鼠标光标从元素外部首次移动到元素范围之内时触发。这个事件不冒泡，而且在光标移动到后代元素上不会触发。DOM2 级事件并没有定义这个事件，但 DOM3 级事件将它纳入了规范。IE、Firefox 9+和 Opera 支持这个事件。
- mouseleave：在位于元素上方的鼠标光标移动到元素范围之外时触发。这个事件不冒泡，而且在光标移动到后代元素上不会触发。DOM2 级事件并没有定义这个事件，但 DOM3 级事件将它纳入了规范。IE、Firefox 9+和 Opera 支持这个事件。

- mousemove：当鼠标指针在元素内部移动时重复地触发。不能通过键盘触发这个事件。
- mouseout：在鼠标指针位于一个元素上方，然后用户将其移入另一个元素时触发。又移入的另一个元素可能位于前一个元素的外部，也可能是这个元素的子元素。不能通过键盘触发这个事件。
- mouseover：在鼠标指针位于一个元素外部，然后用户将其首次移入另一个元素边界之内时触发。不能通过键盘触发这个事件。
- mouseup：在用户释放鼠标按钮时触发。不能通过键盘触发这个事件。

17. 页面上的所有元素都支持鼠标事件。

18. 除了 mouseenter 和 mouseleave，所有鼠标事件都会冒泡，也可以被取消，而取消鼠标事件将会影响浏览器的默认行为。取消鼠标事件的默认行为还会影响其他事件，因为鼠标事件与其他事件是密不可分的关系。

19. 只有在主鼠标按钮被单击（或键盘回车键被按下）时才会触发 click 事件。在常规的设置中，主鼠标按钮就是鼠标左键，而次鼠标按钮就是鼠标右键。

20. 有 3 个键盘事件，简述如下。

- keydown：当用户按下键盘上的任意键时触发，而且如果按住不放的话，会重复触发此事件。
- keypress：当用户按下键盘上的字符键时触发，而且如果按住不放的话，会重复触发此事件。

在用户按了一下键盘上的字符键时，首先会触发 keydown 事件，然后紧跟着是 keypress 事件

21. 虽然所有元素都支持以上 3 个事件，但只有在用户通过文本框输入文本时才最常用到。只有一个文本事件：textInput。

22. 在发生 keydown 和 keyup 事件时，event 对象的 keyCode 属性中会包含一个代码，与键盘上一个特定的键对应。对数字字母字符键，keyCode 属性的值与 ASCII 码中对应小写字母或数字的编码相同。

23. “DOM3 级事件”规范中引入了一个新事件，名叫 textInput。根据规范，当用户在可编辑区域中输入字符时，就会触发这个事件。

24. 复合事件（composition event）是 DOM3 级事件中新添加的一类事件，用于处理 IME 的输入序列。IME（Input Method Editor，输入法编辑器）可以让用户输入在物理键盘上找不到的字符。例如，使用拉丁文键盘的用户通过 IME 照样能输入日文字符。（复合事件、物理键盘上找不到的字符）

    注意：复合事件用处不大。

25. DOM2 级的变动（mutation）事件能在 DOM 中的某一部分发生变化时给出提示。DOM2 级定义了如下变动事件。（变动事件）

- DOMSubtreeModified：在 DOM 结构中发生任何变化时触发。这个事件在其他任何事件触发后都会触发。
- DOMNodeInserted：在一个节点作为子节点被插入到另一个节点中时触发。

- DOMNodeRemoved：在节点从其父节点中被移除时触发。
- DOMNodeInsertedIntoDocument：在一个节点被直接插入文档或通过子树间接插入文档之后触发。这个事件在 DOMNodeInserted 之后触发。
- DOMNodeRemovedFromDocument：在一个节点被直接从文档中移除或通过子树间接从文档中移除之前触发。这个事件在 DOMNodeRemoved 之后触发。
- DOMAttrModified：在特性被修改之后触发。
- DOMCharacterDataModified：在文本节点的值发生变化时触发

26. 在使用 removeChild()或 replaceChild()从 DOM 中删除节点时，首先会触发 DOMNodeRemoved 事件。

这个事件的目标（event.target）是被删除的节点，而 event.relatedNode 属性中包含着对目标节点父节点的引用。

在这个事件触发时，节点尚未从其父节点删除，因此其 parentNode 属性仍然指向父节点（与 event.relatedNode 相同）。这个事件会冒泡，因而可以在 DOM 的任何层次上面处理它。

27. 在使用 appendChild()、replaceChild()或 insertBefore()向 DOM 中插入节点时，首先会触发 DOMNodeInserted 事件。

28. 上下文菜单，也就是通过单击鼠标右键调出来的菜单。

29. event.preventDefault()：取消事件的默认动作。

30. contextmenu 事件用于实现自定义上下文菜单以及取消默认的上下文菜单。

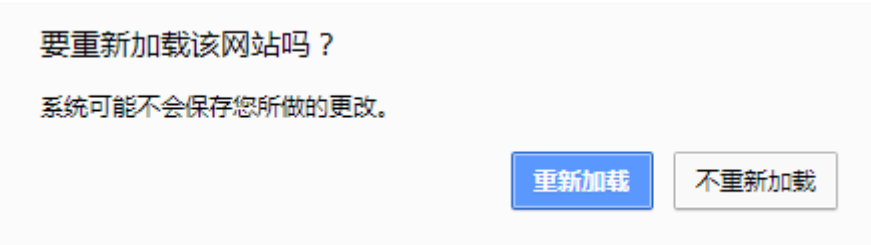
```
EventUtil.addHandler(window, "load",
function(event) {
      var div = document.getElementById("myDiv");
      EventUtil.addHandler(div, "contextmenu",
      function(event) {
            event = EventUtil.getEvent(event);
            EventUtil.preventDefault(event);
            var menu = document.getElementById("myMenu");
            menu.style.left = event.clientX + "px";
            menu.style.top = event.clientY + "px";
            menu.style.visibility = "visible";
      });
      EventUtil.addHandler(document, "click",
      function(event) {
            document.getElementById("myMenu").style.visibility = "hidden";
      });
});
```

EventUtil 的定义如下：

```
var EventUtil = {
```

```
    addHandler: function(element, type, handler){
        if (element.addEventListener){ // 检查传入的元素是否存在 DOM2 级方法
            element.addEventListener(type, handler, false);
        } else if (element.addEvent){ // 如果存在的是 IE 的方法
            element.attachEvent("on" + type, handler); // 则使用 IE 的方法，注意，这里的事件类型必须加上 "on" 前缀。
        } else { // 最后一种可能是使用 DOM0 级
            element["on" + type] = hander;
    }
},

    removeHandler: function(element, type, handler){ // 该方法是删除之前添加的事件处理程序
    if (element.removeEventListener){ // 检查传入的元素是否存在 DOM2 级方法
        element.removeEventListener(type, handler, false); // 若存在，则使用该方法
    } else if (element.detachEvent){ // 如果存在的是 IE 的方法
        element.detachEvent("on" + type, handler); // 则使用 IE 的方法，注意，这里的事件类型必须加上 "on" 前缀。
    } else { // 最后一种可能是使用 DOM0 及方法 (在现代浏览器中，应该不会执行这里的代码)
        element["on" + type] = null;
    }
}
};
```

31. beforeunload 事件：也就是在离开页面时系统为了阻止离开页面而发出的事件。




```
EventUtil.addHandler(window, "beforeunload",
function(event) {
      event = EventUtil.getEvent(event);
      var message = "I'm really going to miss you if you go.";
      event.returnValue = message;
      return message;
});
```

32. DOMContentLoaded 事件则在形成完整的 DOM 树之后就会触发，不理会图像、JavaScript 文件、CSS 文件或其他资源是否已经下载完毕。与 load 事件不同，

DOMContentLoaded 支持在页面下载的早期添加事件处理程序，这也就意味着用户能够尽早地与页面进行交互。

33. Firefox 和 Opera 有一个特性，名叫“往返缓存”（back-forward cache，或 bfcache），可以在用户使用浏览器的“后退”和“前进”按钮时加快页面的转换速度。这个缓存中不仅保存着页面数据，还保存了 DOM 和 JavaScript 的状态；实际上是将整个页面都保存在了内存里。

34. pageshow 事件在页面显示时触发，无论该页面是否来自 bfcache。在重新加载的页面中，pageshow 会在 load 事件触发后触发；而对于 bfcache 中的页面，pageshow 会在页面状态完全恢复的那一刻触发。另外要注意的是，虽然这个事件的目标是 document，但必须将其事件处理程序添加到 window。

```
(function() {
      var showCount = 0;
      EventUtil.addHandler(window, "load",
      function() {
            alert("Load fired");
      });
      EventUtil.addHandler(window, "pageshow",
      function() {
            showCount++;
            alert("Show has been fired " + showCount + " times.");
      });
})();
```

这个例子使用了私有作用域，以防止变量 showCount 进入全局作用域。当页面首次加载完成时，showCount 的值为 0。此后，每当触发 pageshow 事件，showCount 的值就会递增并通过警告框显示出来。如果你在离开包含以上代码的页面之后，又单击“后退”按钮返回该页面，就会看到 showCount 每次递增的值。这是因为该变量的状态，乃至整个页面的状态，都被保存在了内存中，当你返回这个页面时，它们的状态得到了恢复。

35. 与 pageshow 事件对应的是 pagehide 事件，该事件会在浏览器卸载页面的时候触发，而且是在 unload 事件之前触发。

36. HTML5 新增了 hashchange 事件，以便在 URL 的参数列表（及 URL 中“#”号后面的所有字符串）发生变化时通知开发人员。

必须要把 hashchange 事件处理程序添加给 window 对象，然后 URL 参数列表只要变化就会调用它。此时的 event 对象应该额外包含两个属性：oldURL 和 newURL。这两个属性分别保存着参数列表变化前后的完整 URL。例如：

```
EventUtil.addHandler(window, "hashchange", function(event){
    alert("Old URL: " + event.oldURL + "\nNew URL: " + event.newURL);
});
```

37. 触摸事件会在用户手指放在屏幕上面时、在屏幕上滑动时或从屏幕上移开时触发。具体来说，有以下几个触摸事件。

- touchstart：当手指触摸屏幕时触发；即使已经有一个手指放在了屏幕上也会触发。

- touchmove：当手指在屏幕上滑动时连续地触发。在这个事件发生期间，调用preventDefault()可以阻止滚动。
- touchend：当手指从屏幕上移开时触发。
- touchcancel：当系统停止跟踪触摸时触发。

上面这几个事件都会冒泡，也都可以取消。

38. 除了常见的 DOM 属性外，触摸事件还包含下列三个用于跟踪触摸的属性。

(1) touches：表示当前跟踪的触摸操作的 Touch 对象的数组。

(2) targetTouchs：特定于事件目标的 Touch 对象的数组。

(3) changeTouches：表示自上次触摸以来发生了什么改变的 Touch 对象的数组。

39. 每个 Touch 对象包含下列属性：

(1) clientX：触摸目标在视口中的 x 坐标。

(2) clientY：触摸目标在视口中的 y 坐标。

(3) identifier：标识触摸的唯一 ID。

(4) pageX：触摸目标在页面中的 x 坐标。

(5) pageY：触摸目标在页面中的 y 坐标。

(6) screenX：触摸目标在屏幕中的 x 坐标。

(7) screenY：触摸目标在屏幕中的 y 坐标。

(8) target：触摸的 DOM 节点目标。

40. 事件委托可以解决对“事件处理程序过多”的问题。事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。例如，click 事件会一直冒泡到 document 层次。也就是说，我们可以为整个页面指定一个 onclick 事件处理程序，而不必给每个可单击的元素分别添加事件处理程序。以下面的 HTML 代码为例。

```
<ul id="myLinks">
<li id="goSomewhere">Go somewhere</li>
<li id="doSomething">Do something</li>
<li id="sayHi">Say hi</li>
</ul>
```

其中包含 3 个被单击后会执行操作的列表项。

```
var list = document.getElementById("myLinks");
EventUtil.addHandler(list, "click",
function(event) {
      event = EventUtil.getEvent(event);
      var target = EventUtil.getTarget(event);
      switch (target.id) {
      case "doSomething":
            document.title = "I changed the document's title";
            break;
```

```
      case "goSomewhere":
            location.href = "http://www.wrox.com";
            break;
      case "sayHi":
            alert("hi");
            break;
      }
});
```

41. 移除事件处理程序：

```
<div id="myDiv">
      <input type="button" value="Click Me" id="myBtn">
</div>
<script type="text/javascript">
      var btn = document.getElementById("myBtn");
      btn.onclick = function() {
            //先执行某些操作
            btn.onclick = null; //移除事件处理程序
            document.getElementById("myDiv").innerHTML = "Processing...";
      };
</script>
```

42. 模拟触发事件可以用于测试 WEB 应用程序。

43. 可以在 document 对象上使用 createEvent()方法创建 event 对象。

44. 创建新的鼠标事件对象并为其指定必要的信息，就可以模拟鼠标事件。创建鼠标事件对象的方法是为 createEvent()传入字符串"MouseEvents"。返回的对象有一个名为 initMouseEvent()方法，用于指定与该鼠标事件有关的信息。

```
var btn = document.getElementById("myBtn");
//创建事件对象
var event = document.createEvent("MouseEvents");
//初始化事件对象
event.initMouseEvent("click", true, true, document.defaultView, 0, 0, 0, 0, 0,
false, false, false, false, 0, null);
//触发事件
btn.dispatchEvent(event);
```

45. DOM3 级规定，调用 createEvent()并传入"KeyboardEvent"就可以创建一个键盘事件。返回的事件对象会包含一个 initKeyEvent()方法：

```
var textbox = document.getElementById("myTextbox"),
event;
//以 DOM3 级方式创建事件对象
if (document.implementation.hasFeature("KeyboardEvents", "3.0")) {
```

```
    event = document.createEvent("KeyboardEvent");
    //初始化事件对象
    event.initKeyboardEvent("keydown", true, true, document.defaultView, "a",
0, "Shift", 0);
}
//触发事件
textbox.dispatchEvent(event);
```

这个例子模拟的是按住 Shift 的同时又按下 A 键。

46. 要模拟变动事件， 可以使用 createEvent("MutationEvents") 创建一个包含 initMutationEvent() 方法的变动事件对象。

```
var event = document.createEvent("MutationEvents");
event.initMutationEvent("DOMNodeInserted", true, false, someNode, "","","",0);
targ et.dispatchEvent(event);
```

以上代码模拟了 DOMNodeInserted 事件。

第 14 章 表单脚本

1. 在 HTML 中，表单是由<form>元素来表示的，而在 JavaScript 中，表单对应的则是 HTMLFormElement 类型。HTMLFormElement 继承了 HTMLElement，因而与其他 HTML 元素具有相同的默认属性。不过，HTMLFormElement 也有它自己下列独有的属性和方法。

- acceptCharset：服务器能够处理的字符集；等价于 HTML 中的 accept-charset 特性。
- action：接受请求的 URL；等价于 HTML 中的 action 特性。
- elements：表单中所有控件的集合（HTMLCollection）。
- enctype：请求的编码类型；等价于 HTML 中的 enctype 特性。
- length：表单中控件的数量。
- method：要发送的 HTTP 请求类型，通常是"get"或"post"；等价于 HTML 的 method 特性。
- name：表单的名称；等价于 HTML 的 name 特性。
- reset()：将所有表单域重置为默认值。
- submit()：提交表单。
- target：用于发送请求和接收响应的窗口名称；等价于 HTML 的 target 特性。

2. 取得<form>元素引用的方式有好几种：

(1) 最常见的方式就是将它看成与其他元素一样，并为其添加 id 特性，然后再像下面这样使用 getElementById()方法找到它。

```
var form = document.getElementById("form1");
```

(2) 其次，通过 document.forms 可以取得页面中所有的表单。

```
var firstForm = document.forms[0]; //取得页面中的第一个表单
var myForm = document.forms["form2"]; //取得页面中名称为"form2"的表单
```

3. 使用<input>或<button>都可以定义提交按钮，只要将其 type 特性的值设置为"submit"即可，而图像按钮则是通过将<input>的 type 特性值设置为"image"来定义的。因此，只要我们单击以下代码生成的按钮，就可以提交表单。

```
<!-- 通用提交按钮 -->
<input type="submit" value="Submit Form">
<!-- 自定义提交按钮 -->
<button type="submit">Submit Form</button>
<!-- 图像按钮 -->
<input type="image" src="graphic.gif">
```

4. 取消 submit 事件可以取消表单提交。

```
var form = document.getElementById("myForm");
EventUtil.addHandler(form, "submit",
function(event) {
      //取得事件对象
      event = EventUtil.getEvent(event);
      //阻止默认事件
      EventUtil.preventDefault(event);
});
```

5. 在用户单击重置按钮时，表单会被重置。使用 type 特性值为"reset"的<input>或<button>都可以创建重置按钮，如下面的例子所示。

```
<!-- 通用重置按钮 -->
<input type="reset" value="Reset Form">
<!-- 自定义重置按钮 -->
<button type="reset">Reset Form</button>
```

6. 每个表单都有 elements 属性，该属性是表单中所有表单元素（字段）的集合。这个 elements 集合是一个有序列表，其中包含着表单中的所有字段，例如<input>、<textarea>、<button>和<fieldset>。每个表单字段在 elements 集合中的顺序，与它们出现在标记中的顺序相同，可以按照位置和 name 特性来访问它们。下面来看一个例子。

```
var form = document.getElementById("form1");
//取得表单中的第一个字段
var field1 = form.elements[0];
//取得名为"textbox1"的字段
var field2 = form.elements["textbox1"];
//取得表单中包含的字段的数量
var fieldCount = form.elements.length;
```

7. 如果有多个表单控件都在使用一个 name（如单选按钮），那么就会返回以该 name 命名的一个 NodeList。
8. 表单字段共有的属性如下。

- disabled：布尔值，表示当前字段是否被禁用。
- form：指向当前字段所属表单的指针；只读。
- name：当前字段的名称。
- readOnly：布尔值，表示当前字段是否只读。
- tabIndex：表示当前字段的切换（tab）序号。
- type：当前字段的类型，如"checkbox"、"radio"，等等。
- value：当前字段将被提交给服务器的值。对文件字段来说，这个属性是只读的，包含着文件在计算机中的路径。

9. 每个表单字段都有两个方法：focus()和 blur()。

- focus()方法用于将浏览器的焦点设置到表单字段，即激活表单字段，使其可以响应键盘事件。
- blur()方法的作用是从元素中移走焦点。

10. 除了支持鼠标、键盘、更改和 HTML 事件之外，所有表单字段都支持下列 3 个事件。

- blur：当前字段失去焦点时触发。
- change：对于<input>和<textarea>元素，在它们失去焦点且 value 值改变时触发；对于<select>元素，在其选项改变时触发。
- focus：当前字段获得焦点时触发。

11. HTML5 为表单字段新增了一个 autofocus 属性。在支持这个属性的浏览器中，只要设置这个属性，不用 JavaScript 就能自动把焦点移动到相应字段。
12. 在 HTML 中，有两种方式来表现文本框：一种是使用<input>元素的单行文本框，另一种是使用<textarea>的多行文本框。
13. 要表现文本框，必须将<input>元素的 type 特性设置为"text"。而通过设置 size 特性，可以指定文本框中能够显示的字符数。通过 value 特性，可以设置文本框的初始值，而 maxlength 特性则用于指定文本框可以接受的最大字符数。如果要创建一个文本框，让它能够显示 25 个字符，但输入不能超过 50 个字符，可以使用以下代码：

```
<input type="text" size="25" maxlength="50" value="initial value">
```

14. <textarea>元素则始终会呈现为一个多行文本框。要指定文本框的大小，可以使用 rows 和 cols 特性。其中，rows 特性指定的是文本框的字符行数，而 cols 特性指定的是文本框的字符列数。
15. 与<input>元素不同，<textarea>的初始值必须要放在<textarea>和</textarea>之间，如下面的例子所示。

```
<textarea rows="25" cols="5">initial value</textarea>
```

16. 无论这两种文本框在标记中有什么区别，但它们都会将用户输入的内容保存在 value 属性中。可以通过这个属性读取和设置文本框的值。

```
var textbox = document.forms[0].elements["textbox1"];
alert(textbox.value);
textbox.value = "Some new value";
```

建议读者像上面这样使用 value 属性读取或设置文本框的值，不建议使用标准的 DOM 方法。

17. select()方法用于选择文本框中的所有文本。

```
var textbox = document.forms[0].elements["textbox1"];
textbox.select();
```

18. 在选择了文本框中的文本时，就会触发 select 事件。

19. HTML5 添加两个属性：selectionStart 和 selectionEnd。这两个属性中保存的是基于 0 的数值，表示所选择文本的范围（即文本选区开头和结尾的偏移量）。因此，要取得用户在文本框中选择的文本，可以使用如下代码。

20. setSelectionRange()方法用于选择文本框中的部分文本。

```
textbox.value = "Hello world!"
//选择所有文本
textbox.setSelectionRange(0, textbox.value.length); //"Hello world!"
//选择前 3 个字符
textbox.setSelectionRange(0, 3); //"Hel"
//选择第 4 到第 6 个字符
textbox.setSelectionRange(4, 7); //"o w"
```

21. 在极端的情况下，可以通过阻止事件的方式屏蔽所有按键操作：

```
EventUtil.addHandler(textbox, "keypress",
function(event) {
      event = EventUtil.getEvent(event);
      EventUtil.preventDefault(event);
});
```

22. 如果只想屏蔽特定的字符，则需要检测 keypress 事件对应的字符编码，然后再决定如何响应。例如，下列代码只允许用户输入数值。

```
EventUtil.addHandler(textbox, "keypress",
function(event) {
      event = EventUtil.getEvent(event);
      var target = EventUtil.getTarget(event);
      var charCode = EventUtil.getCharCode(event);
```

```
      if (!/\d/.test(String.fromCharCode(charCode))) {
            EventUtil.preventDefault(event);
      }
});
```

使用正则表达式 /\d/ 来测试该字符串，从而确定用户输入的是不是数值。如果测试失败，那么就使用 EventUtil.preventDefault()屏蔽按键事件。结果，文本框就会忽略所有输入的非数值。

23. HTML 5 把剪贴板事件纳入了规范。下列就是 6 个剪贴板事件。

- beforecopy：在发生复制操作前触发。
- copy：在发生复制操作时触发。
- beforecut：在发生剪切操作前触发。
- cut：在发生剪切操作时触发。
- beforepaste：在发生粘贴操作前触发。
- paste：在发生粘贴操作时触发。

24. 为增强易用性，同时加快数据输入，可以在前一个文本框中的字符达到最大数量后，自动将焦点切换到下一个文本框。换句话说，用户在第一个文本框中输入了 3 个数字之后，焦点就会切换到第二个文本框，再输入 3 个数字，焦点又会切换到第三个文本框。这种“自动切换焦点”的功能，可以通过下列代码实现：

```
(function() {
      function tabForward(event) {
            event = EventUtil.getEvent(event);
            var target = EventUtil.getTarget(event);
            if (target.value.length == target.maxLength) {
                  var form = target.form;
                  for (var i = 0,
                  len = form.elements.length; i < len; i++) {
                        if (form.elements[i] == target) {
                              if (form.elements[i + 1]) {
                                    form.elements[i + 1].focus();
                              }
                              return;
                        }
                  }
            }
      }
      var textbox1 = document.getElementById("txtTel1");
      var textbox2 = document.getElementById("txtTel2");
      var textbox3 = document.getElementById("txtTel3");
      EventUtil.addHandler(textbox1, "keyup", tabForward);
      EventUtil.addHandler(textbox2, "keyup", tabForward);
      EventUtil.addHandler(textbox3, "keyup", tabForward);
```

```
})();
```

25. HTML5 约束验证：

(1) 必填字段。在表单字段中指定了 required 属性，如下面的例子所示：

```
<input type="text" name="username" required>
```

(2) HTML5 为文本字段新增了 pattern 属性。这个属性的值是一个正则表达式，用于匹配文本框中的值。例如，如果只想允许在文本字段中输入数值，可以像下面的代码一样应用约束：

```
<input type="text" pattern="\d+" name="count">
```

注意，模式的开头和末尾不用加^和$符号（假定已经有了）。这两个符号表示输入的值必须从头到尾都与模式匹配。

(3) 使用 checkValidity()方法可以检测表单中的某个字段是否有效。所有表单字段都有个方法，如果字段的值有效，这个方法返回 true，否则返回 false。

```
if (document.forms[0].elements[0].checkValidity()) {
    //字段有效，继续
} else {
    //字段无效
}
```

(4) 与 checkValidity()方法简单地告诉你字段是否有效相比，validity 属性则会告诉你为什么字段有效或无效。这个对象中包含一系列属性，每个属性会返回一个布尔值。

- customError ：如果设置了 setCustomValidity()，则为 true，否则返回 false。
- patternMismatch：如果值与指定的 pattern 属性不匹配，返回 true。
- rangeOverflow：如果值比 max 值大，返回 true。
- rangeUnderflow：如果值比 min 值小，返回 true。
- stepMisMatch：如果 min 和 max 之间的步长值不合理，返回 true。
- tooLong：如果值的长度超过了 maxlength 属性指定的长度，返回 true。有的浏览器（如 Firefox 4）会自动约束字符数量，因此这个值可能永远都返回 false。
- typeMismatch：如果值不是"mail"或"url"要求的格式，返回 true。
- valid：如果这里的其他属性都是 false，返回 true。checkValidity()也要求相同的值。
- valueMissing：如果标注为 required 的字段中没有值，返回 true。

(5) 通过设置 novalidate 属性，可以告诉表单不进行验证。

```
<form method="post" action="signup.php" novalidate>
    <!--这里插入表单元素-->
</form>
```

26. 选择框是通过<select>和<option>元素创建的。为了方便与这个控件交互，除了所有表单字段共有的属性和方法外，HTMLSelectElement 类型还提供了下列属性和方法。

- add(newOption, relOption)：向控件中插入新<option>元素，其位置在相关项（relOption）之前。
- multiple：布尔值，表示是否允许多项选择；等价于 HTML 中的 multiple 特性。
- options：控件中所有<option>元素的 HTMLCollection。
- remove(index)：移除给定位置的选项。
- selectedIndex：基于 0 的选中项的索引，如果没有选中项，则值为-1。对于支持多选的控件，只保存选中项中第一项的索引。
- size：选择框中可见的行数；等价于 HTML 中的 size 特性。

27. 选择框的 type 属性不是"select-one"，就是"select-multiple"，这取决于 HTML 代码中有没有 multiple 特性。

28. 对于只允许选择一项的选择框，访问选中项的最简单方式，就是使用选择框的 selectedIndex 属性，如下面的例子所示：

```
var selectedOption = selectbox.options[selectbox.selectedIndex];
```

29. 对于可以选择多项的选择框，selectedfIndex 属性就好像只允许选择一项一样。设置 selectedIndex 会导致取消以前的所有选项并选择指定的那一项，而读取 selectedIndex 则只会返回选中项中第一项的索引值。

30. selected 属性的作用主要是确定用户选择了选择框中的哪一项。要取得所有选中的项，可以循环遍历选项集合，然后测试每个选项的 selected 属性。来看下面的例子。

```
function getSelectedOptions(selectbox) {
      var result = new Array();
      var option = null;
      for (var i = 0,len = selectbox.options.length; i < len; i++) {
            option = selectbox.options[i];
            if (option.selected) {
                  result.push(option);
            }
      }
      return result;
}
```

31. 添加选项的方式有很多：

(1) 第一种方式就是使用如下所示的 DOM 方法。

```
var newOption = document.createElement("option");
newOption.appendChild(document.createTextNode("Option text"));
newOption.setAttribute("value", "Option value");
selectbox.appendChild(newOption);
```

(2) 第二种方式是使用 Option 构造函数来创建新选项

```
var newOption = new Option("Option text", "Option value");
selectbox.appendChild(newOption); //在 IE8 及之前版本中有问题
```

(3) 第三种添加新选项的方式是使用选择框的 add()方法。

```
var newOption = new Option("Option text", "Option value");
selectbox.add(newOption, undefined); //最佳方案
```

32. 与添加选项类似，移除选项的方式也有很多种。

(1) 首先，可以使用 DOM 的 removeChild()方法，为其传入要移除的选项，如下面的例子所示：

```
selectbox.removeChild(selectbox.options[0]); //移除第一个选项
```

(2) 其次，可以使用选择框的 remove()方法。

```
selectbox.remove(0); //移除第一个选项
```

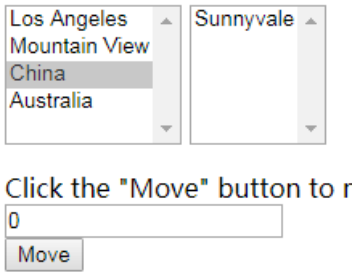
(3) 最后一种方式，就是将相应选项设置为 null。

```
selectbox.options[0] = null; //移除第一个选项
```

33. DOM 的 appendChild()方法可以将第一个选择框中的选项直接移动到第二个选择框中。

```
var selectbox1 = document.getElementById("selLocations1");
var selectbox2 = document.getElementById("selLocations2");
selectbox2.appendChild(selectbox1.options[0]);
```

该方法的作用也就类似下图：



34. 富文本技术的本质，就是在页面中嵌入一个包含空 HTML 页面的 iframe。通过设置 designMode 属性，这个空白的 HTML 页面可以被编辑，而编辑对象则是该页面<body>元素的 HTML 代码。

```
<iframe name="richedit" style="height:100px;width:100px;" src="blank.htm">
</iframe>
<script type="text/javascript">
      EventUtil.addHandler(window, "load",
      function() {
```

```
            frames["richedit"].document.designMode = "on";
      });
</script>
```

只有在页面完全加载之后才能设置这个属性。因此，在包含页面中，需要使用 onload 事件处理程序来在恰当的时刻设置 designMode。

35. 另一种编辑富文本内容的方式是使用名为 contenteditable 的特殊属性：

```
<div class="editable" id="richedit" contenteditable></div>
```

这样，元素中包含的任何文本内容就都可以编辑了，就好像这个元素变成了<textarea>元素一样。

通过在这个元素上设置 contenteditable 属性，也能打开或关闭编辑模式。

```
var div = document.getElementById("richedit");
div.contentEditable = "true";
```

36. 与富文本编辑器交互的主要方式，就是使用 document.execCommand()。这个方法可以对文档执行预定义的命令，而且可以应用大多数格式。
37. 在富文本编辑器中，使用框架（iframe）的 getSelection()方法，可以确定实际选择的文本。
38. 由于富文本编辑是使用 iframe 而非表单控件实现的，因此从技术上说，富文本编辑器并不属于表单。换句话说，富文本编辑器中的 HTML 不会被自动提交给服务器，而需要我们手工来提取并提交 HTML。

为此，通常可以添加一个隐藏的表单字段，让它的值等于从 iframe 中提取出的 HTML。具体来说，就是在提交表单之前，从 iframe 中提取出 HTML，并将其插入到隐藏的字段中。下面就是通过表单的 onsubmit 事件处理程序实现上述操作的代码。

```
EventUtil.addHandler(form, "submit",
function(event) {
      event = EventUtil.getEvent(event);
      var target = EventUtil.getTarget(event);
      target.elements["comments"].value =
frames["richedit"].document.body.innerHTML;
});
```

在此，我们通过文档主体的 innerHTML 属性取得了 iframe 中的 HTML，然后将其插入到了名为"comments"的表单字段中。这样可以确保恰好在提交表单之前填充"comments"字段。

第十五章 使用 **Canvas** 绘图

1. <canvas>元素负责在页面中设定一个区域，然后就可以通过 JavaScript 动态地在这个区域中绘制图形。
2. 要使用<canvas>元素，必须先设置其 width 和 height 属性，指定可以绘图的区域大小。出现在开始和结束标签中的内容是后备信息，如果浏览器不支持<canvas>元素，就会

显示这些信息。下面就是<canvas>元素的例子。

```
<canvas id="drawing" width=" 200" height="200">A drawing of something.</canvas>
```

3. 要在这块画布（canvas）上绘图，需要取得绘图上下文。而取得绘图上下文对象的引用，需要调用 getContext()方法并传入上下文的名字。传入"2d"，就可以取得 2D 上下文对象。

```
var drawing = document.getElementById("drawing");
//确定浏览器支持<canvas>元素
if (drawing.getContext){
    var context = drawing.getContext("2d");
//更多代码
}
```

在使用<canvas>元素之前，首先要检测 getContext()方法是否存在，这一步非常重要。

4. 使用 toDataURL()方法，可以导出在<canvas>元素上绘制的图像。比如，要取得画布中的一幅 PNG 格式的图像，可以使用以下代码。

```
var drawing = document.getElementById("drawing");
//确定浏览器支持<canvas>元素
if (drawing.getContext){
      //取得图像的数据 URI
      var imgURI = drawing.toDataURL("image/png");
      //显示图像
      var image = document.createElement("img");
      image.src = imgURI;
      document.body.appendChild(image);
}
```

5. 2D 上下文的两种基本绘图操作是填充和描边。

- 填充，就是用指定的样式（颜色、渐变或图像）填充图形；
- 描边，就是只在图形的边缘画线。

6. 矩形是唯一一种可以直接在 2D 上下文中绘制的形状。与矩形有关的方法包括 fillRect()、strokeRect()和 clearRect()。

- fillRect()方法在画布上绘制的矩形会填充指定的颜色。填充的颜色通过 fillStyle 属性指定。

```
var drawing = document.getElementById("drawing");
//确定浏览器支持<canvas>元素
if (drawing.getContext) {
      var context = drawing.getContext("2d");
/*
* 根据 Mozilla 的文档
* http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
```

```
*/
      //绘制红色矩形
      context.fillStyle = "#ff0000";
      context.fillRect(10, 10, 50, 50);
      //绘制半透明的蓝色矩形
      context.fillStyle = "rgba(0,0,255,0.5)";
      context.fillRect(30, 30, 50, 50);
}
```

- strokeRect()方法在画布上绘制的矩形会使用指定的颜色描边。描边颜色通过 strokeStyle 属性指定。
- clearRect()方法用于清除画布上的矩形区域。本质上，这个方法可以把绘制上下文中的某一矩形区域变透明。

7. 2D 绘制上下文支持很多在画布上绘制路径的方法。通过路径可以创造出复杂的形状和线条。要绘制路径，首先必须调用 beginPath()方法，表示要开始绘制新路径。然后，再通过调用下列方法来实际地绘制路径。

(1) arc(x, y, radius, startAngle, endAngle, counterclockwise)：以(x,y)为圆心绘制一条弧线，弧线半径为 radius，起始和结束角度（用弧度表示）分别为 startAngle 和 endAngle。最后一个参数表示 startAngle 和 endAngle 是否按逆时针方向计算，值为 false 表示按顺时针方向计算。

(2) arcTo(x1, y1, x2, y2, radius)：从上一点开始绘制一条弧线，到(x2,y2)为止，并且以给定的半径 radius 穿过(x1,y1)。

(3) bezierCurveTo(c1x, c1y, c2x, c2y, x, y)：从上一点开始绘制一条曲线，到(x,y)为止，并且以(c1x,c1y)和(c2x,c2y)为控制点。

(4) lineTo(x, y)：从上一点开始绘制一条直线，到(x,y)为止。

(5) moveTo(x, y)：将绘图游标移动到(x,y)，不画线。

(6) quadraticCurveTo(cx, cy, x, y)：从上一点开始绘制一条二次曲线，到(x,y)为止，并且以(cx,cy)作为控制点。

(7) rect(x, y, width, height)：从点(x,y)开始绘制一个矩形，宽度和高度分别由 width 和 height 指定。这个方法绘制的是矩形路径，而不是 strokeRect()和 fillRect()所绘制的独立的形状。

8. 如果想绘制一条连接到路径起点的线条，可以调用 closePath()。如果路径已经完成，你想用 fillStyle 填充它，可以调用 fill()方法。另外，还可以调用 stroke()方法对路径描边，描边使用的是 strokeStyle。最后还可以调用 clip()，这个方法可以在路径上创建一个剪切区域。

9. 绘制文本主要有两个方法：fillText()和 strokeText()。这两个方法都以下列 3 个属性为基础。

(1) font：表示文本样式、大小及字体，用 CSS 中指定字体的格式来指定，例如"10px Arial"。

(2) textAlign：表示文本对齐方式。可能的值有"start"、"end"、"left"、"right"和"center"。建议使用"start"和"end"，不要使用"left"和"right"，因为前两者的意思更稳妥，能同时适合从左到右和从右到左显示（阅读）的语言。

(3) textBaseline：表示文本的基线。可能的值有"top"、"hanging"、"middle"、"alphabetic"、"ideographic"和"bottom"。

10. fillText()方法使用 fillStyle 属性绘制文本，而 strokeText()方法使用 strokeStyle 属性为文本描边。

11. 2D 绘制上下文支持各种基本的绘制变换。可以通过如下方法来修改变换矩阵。

- rotate(angle)：围绕原点旋转图像 angle 弧度。
- scale(scaleX, scaleY)：缩放图像，在 x 方向乘以 scaleX，在 y 方向乘以 scaleY。scaleX 和 scaleY 的默认值都是 1.0。
- translate(x, y)：将坐标原点移动到(x,y)。执行这个变换之后，坐标(0,0)会变成之前由(x,y)表示的点。
- transform(m1_1, m1_2, m2_1, m2_2, dx, dy)：直接修改变换矩阵，方式是乘以如下矩阵。
- setTransform(m1_1, m1_2, m2_1, m2_2, dx, dy)：将变换矩阵重置为默认状态，然后再调用 transform()

12. 2D 绘图上下文内置了对图像的支持。如果你想把一幅图像绘制到画布上，可以使用 drawImage()方法。

```
var image = document.images[0];
context.drawImage(image, 10, 10);
```

13. 2D 上下文会根据以下几个属性的值，自动为形状或路径绘制出阴影。

- shadowColor：用 CSS 颜色格式表示的阴影颜色，默认为黑色。
- shadowOffsetX：形状或路径 x 轴方向的阴影偏移量，默认为 0。
- shadowOffsetY：形状或路径 y 轴方向的阴影偏移量，默认为 0。
- shadowBlur：模糊的像素数，默认 0，即不模糊。

这些属性都可以通过 context 对象来修改。只要在绘制前为它们设置适当的值，就能自动产生阴影。例如：

```
var context = drawing.getContext("2d");
//设置阴影
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur = 4;
context.shadowColor = "rgba(0, 0, 0, 0.5)";
//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
```

```
//绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

14. 渐变由 CanvasGradient 实例表示，很容易通过 2D 上下文来创建和修改。要创建一个新的线性渐变，可以调用 createLinearGradient()方法。
15. 创建了渐变对象后，下一步就是使用 addColorStop()方法来指定色标。

```
var gradient = context.createLinearGradient(30, 30, 70, 70);
gradient.addColorStop(0, "white");
gradient.addColorStop(1, "black");
```

16. 模式其实就是重复的图像， 可以用来填充或描边图形。要创建一个新模式， 可以调用 createPattern()方法。
17. 2D 上下文的一个明显的长处就是，可以通过 getImageData()取得原始图像数据。

```
var imageData = context.getImageData(10, 5, 50, 50);
```

18. globalAlpha 属性 用于指定所有绘制的透明度，是一个介于 0 和 1 之间的值（包括 0 和 1），默认值为 0。如果所有后续操作都要基于相同的透明度，就可以先把 globalAlpha 设置为适当值，然后绘制，最后再把它设置回默认值 0。

```
//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
//修改全局透明度
context.globalAlpha = 0.5;
//绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
//重置全局透明度
context.globalAlpha = 0;
```

19. globalCompositionOperation 属性表示后绘制的图形怎样与先绘制的图形结合。
20. WebGL 是针对 Canvas 的 3D 上下文。

第 16 章 HTML5 脚本编程

1. 跨文档消息传送，有时候简称为 XDM，指的是在来自不同域的页面间传递消息。例如，www.wrox.com 域中的页面与位于一个内嵌框架中的 p2p.wrox.com 域中的页面通信。
2. XDM 的核心是 postMessage()方法，用于向另一个地方传递数据。对于 XDM 而言，“另一个地方”指的是包含在当前页面中的<iframe>元素，或者由当前页面弹出的窗

口。

```
//注意：所有支持 XDM 的浏览器也支持 iframe 的 contentWindow 属性
var iframeWindow = document.getElementById("myframe").contentWindow;
iframeWindow.postMessage("A secret", "http://www.wrox.com");
```

3. 通过拖放事件，可以控制拖放相关的各个方面。拖动某元素时，将依次触发下列事件：

- dragstart
- drag
- dragend

4. 当某个元素被拖动到一个有效的放置目标上时，下列事件会依次发生：

- dragenter：只要有元素被拖动到放置目标上，就会触发 dragenter 事件。
- dragover：被拖动的元素还在放置目标的范围内移动时，就会持续触发 dragover 事件。
- dragleave 或 drop：如果元素被拖出了放置目标，会触发 dragleave 事件。如果元素被放到了放置目标中，则会触发 drop 事件而不是 dragleave 事件。

5. 在拖动元素经过某些无效放置目标时，可以看到一种特殊的光标，表示不能放置。你可以把任何元素变成有效的放置目标，方法是重写 dragenter 和 dragover 事件的默认行为。

```
var droptarget = document.getElementById("droptarget");
EventUtil.addHandler(droptarget, "dragover",
function(event) {
      EventUtil.preventDefault(event);
});
EventUtil.addHandler(droptarget, "dragenter",
function(event) {
      EventUtil.preventDefault(event);
});
```

以上代码执行后，你就会发现当拖动着元素移动到放置目标上时，光标变成了允许放置的符号。当然，释放鼠标也会触发 drop 事件。

6. dataTransfer 对象，它是事件对象的一个属性，用于从被拖动元素向放置目标传递字符串格式的数据。因为它是事件对象的属性，所以只能在拖放事件的事件处理程序中访问 dataTransfer 对象。

7. dataTransfer 对象有两个主要方法：getData()和 setData()。

- getData()获取数据。
- setData()保存数据。

```
//设置和接收文本数据
event.dataTransfer.setData("text", "some text");
var text = event.dataTransfer.getData("text");
//设置和接收 URL
```

```
event.dataTransfer.setData("URL", "http://www.wrox.com/");
var url = event.dataTransfer.getData("URL");
```

8. 保存在 dataTransfer 对象中的数据只能在 drop 事件处理程序中读取。如果在 ondrop 处理程序中没有读到数据，那就是 dataTransfer 对象已经被销毁，数据也丢失了。
9. 通过 dataTransfer 对象的 dropEffect 属性可以知道被拖动的元素能够执行哪种放置行为。这个属性有下列 4 个可能的值。

- "none"：不能把拖动的元素放在这里。这是除文本框之外所有元素的默认值。
- "move"：应该把拖动的元素移动到放置目标。
- "copy"：应该把拖动的元素复制到放置目标。
- "link"：表示放置目标会打开拖动的元素（但拖动的元素必须是一个链接，有 URL）。

10. dropEffect 属性只有搭配 effectAllowed 属性才有用。effectAllowed 属性表示允许拖动元素的哪种 dropEffect。

- effectAllowed 属性可能的值如下。
- "uninitialized"：没有给被拖动的元素设置任何放置行为。
- "none"：被拖动的元素不能有任何行为。
- "copy"：只允许值为"copy"的 dropEffect。
- "link"：只允许值为"link"的 dropEffect。
- "move"：只允许值为"move"的 dropEffect。
- "copyLink"：允许值为"copy"和"link"的 dropEffect。
- "copyMove"：允许值为"copy"和"move"的 dropEffect。
- "linkMove"：允许值为"link"和"move"的 dropEffect。
- "all"：允许任意 dropEffect

11. draggable 属性，表示元素是否可以拖动。图像和链接的 draggable 属性自动被设置成了 true，而其他元素这个属性的默认值都是 false。要想让其他元素可拖动，或者让图像或链接不能拖动，都可以设置这个属性。

```
<!-- 让这个图像不可以拖动 -->
<img src="smile.gif" draggable="false" alt="Smiley face">
<!-- 让这个元素可以拖动 -->
<div draggable="true">...</div>
```

12. HTML5 规范规定 dataTransfer 对象还应该包含下列方法和属性。

- addElement(element)：为拖动操作添加一个元素。添加这个元素只影响数据（即增加作为拖动源而响应回调的对象），不会影响拖动操作时页面元素的外观。
- clearData(format)：清除以特定格式保存的数据。

- setDragImage(element, x, y)：指定一幅图像，当拖动发生时，显示在光标下方。这个方法接收的三个参数分别是要显示的 HTML 元素和光标在图像中的 x、y 坐标。其中，HTML 元素可以是一幅图像，也可以是其他元素。是图像则显示图像，是其他元素则显示渲染后的元素。
- types：当前保存的数据类型。这是一个类似数组的集合，以"text"这样的字符串形式保存着数据类型。

13. HTML5 使用<audio>标签插入音频，使用<video>标签插入视频：

```
<!-- 嵌入视频 -->
<video src="conference.mpg" id="myVideo">Video player not available.</video>
<!-- 嵌入音频 -->
<audio src="song.mp3" id="myAudio">Audio player not available.</audio>
```

使用这两个元素时，至少要在标签中包含 src 属性，指向要加载的媒体文件。

14. 因为并非所有浏览器都支持所有媒体格式，所以可以指定多个不同的媒体来源。为此，不用在标签中指定 src 属性，而是要像下面这样使用一或多个<source>元素。

```
<!-- 嵌入视频 -->
<video id="myVideo">
<source src="conference.webm" type="video/webm; codecs='vp8, vorbis'">
<source src="conference.ogv" type="video/ogg; codecs='theora, vorbis'">
<source src="conference.mpg">
Video player not available.
</video>
<!-- 嵌入音频 -->
<audio id="myAudio">
<source src="song.ogg" type="audio/ogg">
<source src="song.mp3" type="audio/mpeg">
Audio player not available.
</audio>
```

第 17 章 错误处理与调试

1. ECMA-262 第 3 版引入了 try-catch 语句，基本的语法如下所示：

```
try {
      // 可能会导致错误的代码
} catch(error) {
      // 在错误发生时怎么处理
}
```

2. finally 子句：虽然在 try-catch 语句中是可选的，但 finally 子句一经使用，其代码无论如何都会执行。换句话说，try 语句块中的代码全部正常执行，finally 子句会执行；如果因为出错而执行了 catch 语句块，finally 子句照样还会执行。

```
function testFinally() {
      try {
            return 2;
      } catch(error) {
            return 1;
      } finally {
            return 0;
      }
}
```

3. 与 try-catch 语句相配的还有一个 throw 操作符，用于随时抛出自定义错误。抛出错误时，必须要给 throw 操作符指定一个值，这个值是什么类型，没有要求。下列代码都是有效的。

```
throw 12345;
throw "Hello world!";
throw true;
throw { name: "JavaScript"};
throw new Error("Something bad happened.");
```

4. ECMA-262 定义了下列 7 种错误类型：

- Error：基类型，其他错误类型都继承自该类型。
- EvalError：EvalError 类型的错误会在使用 eval()函数而发生异常时被抛出。
- RangeError：RangeError 类型的错误会在数值超出相应范围时触发。
- ReferenceError：在找不到对象的情况下，会触发 ReferenceError。
- SyntaxError：把语法错误的 JavaScript 字符串传入 eval()函数时，就会导致此类错误。
- TypeError：在变量中保存着意外的类型时，或者在访问不存在的方法时，都会导致这种错误。
- URIError：URI 格式不正确时，就会导致 URIError 错误。

5. 常见的错误类型：

- 类型转换错误
- 数据类型错误
- 通信错误

6. JavaScript 是松散类型的，为了保证不会发生数据类型错误，只能依靠开发人员编写适当的数据类型检测代码。

```
//不安全的函数，任何非字符串值都会导致错误
function getQueryString(url) {
      var pos = url.indexOf("?");
      if (pos > -1) {
```

```
            return url.substring(pos + 1);
      }
      return "";
}
```

这个例子中的两个函数只能操作字符串，因此只要传入其他数据类型的值就会导致错误。而添加一条简单的类型检测语句，就可以确保函数不那么容易出错。

```
function getQueryString(url) {
      if (typeof url == "string") { //通过检查类型确保安全
            var pos = url.indexOf("?");
            if (pos > -1) {
                  return url.substring(pos + 1);
            }
      }
      return "";
}
```

7. 开发 Web 应用程序过程中的一种常见的做法，就是集中保存错误日志，以便查找重要错误的原因。

第 18 章 JavaScript 与 XML

1. 在支持 DOM2 级的浏览器中使用以下语法来创建一个空白的 XML 文档：

```
var xmldom = document.implementation.createDocument(namespaceUri, root,
doctype);
```

因此，要想创建一个新的、文档元素为<root>的 XML 文档，可以使用如下代码：

```
var xmldom = document.implementation.createDocument("", "root", null);
alert(xmldom.documentElement.tagName); //"root"
var child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

2. DOMParser 类型用于将 XML 解析为 DOM 文档。在解析 XML 之前，首先必须创建一个 DOMParser 的实例，然后再调用 parseFromString()方法。

```
var parser = new DOMParser();
var xmldom = parser.parseFromString("<root><child/></root>", "text/xml");
alert(xmldom.documentElement.tagName); //"root"
alert(xmldom.documentElement.firstChild.tagName); //"child"
var anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);
var children = xmldom.getElementsByTagName("child");
alert(children.length); //2
```

3. XMLSerializer 类型用于将 DOM 文档解释成 XML 文件。要序列化 DOM 文档，首先

必须创建 XMLSerializer 的实例，然后将文档传入其 serializeTo-String ()方法：

```
var serializer = new XMLSerializer();
var xml = serializer.serializeToString(xmldom);
alert(xml);
```

4. XPath 是设计用来在 DOM 文档中查找节点的一种手段。
5. 要确定某浏览器是否支持 DOM3 级 XPath，可以使用以下 JavaScript 代码：

```
var supportsXPath = document.implementation.hasFeature("XPath", "3.0");
```

6. XPathEvaluator 类型用于在特定的上下文中对 XPath 表达式求值。这个类型有下列 3 个方法。

- createExpression(expression, nsresolver)：将 XPath 表达式及相应的命名空间信息转换成一个 XPathExpression，这是查询的编译版。在多次使用同一个查询时很有用。
- createNSResolver(node)：根据 node 的命名空间信息创建一个新的 XPathNSResolver 对象。在基于使用命名空间的 XML 文档求值时，需要使用 XPathNSResolver 对象。
- evaluate(expression, context, nsresolver, type, result)：在给定的上下文中，基于特定的命名空间信息来对 XPath 表达式求值。剩下的参数指定如何返回结果。

```
var result = xmldom.evaluate("employee/name", xmldom.documentElement, null,
XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);
if (result !== null) {
      var node = result.iterateNext();
      while (node) {
            alert(node.tagName);
            node = node.iterateNext();
      }
}
```

如果没有节点匹配 XPath 表达式，evaluate()返回 null；否则，它会返回一个 XPathResult 对象。这个 XPathResult 对象带有的属性和方法，可以用来取得特定类型的结果。如果节点是一个节点迭代器，无论是次序一致还是次序不一致的，都必须要使用 iterateNext()方法从节点中取得匹配的节点。

第二十章 JSON

1. 关于 JSON，最重要的是要理解它是一种数据格式，不是一种编程语言。
2. JSON 的语法可以表示以下三种类型的值。

- 简单值：使用与 JavaScript 相同的语法，可以在 JSON 中表示字符串、数值、布尔值和 null。但 JSON 不支持 JavaScript 中的特殊值 undefined。
- 对象：对象作为一种复杂数据类型，表示的是一组无序的键值对儿。而每个键值对儿中的值可以是简单值，也可以是复杂数据类型的值。

- 数组：数组也是一种复杂数据类型，表示一组有序的值的列表，可以通过数值索引来访问其中的值。数组的值也可以是任意类型——简单值、对象或数组。

3. JSON 不支持变量、函数或对象实例。
4. 最简单的 JSON 数据形式就是简单值。

```
"Hello world!"
```

JavaScript 字符串与 JSON 字符串的最大区别在于，JSON 字符串必须使用双引号，单引号会导致语法错误。

5. JSON 中的对象与 JavaScript 字面量稍微有一些不同。下面是一个 JavaScript 中的对象字面量：

```
var person = {
      name: "Nicholas",
      age: 29
};
```

JSON 中的对象要求给属性加引号：

```
var object = {
      "name": "Nicholas",
      "age": 29
};
```

JSON 表示上述对象的方式如下：

```
{
      "name": "Nicholas",
      "age": 29
}
```

6. JSON 中的第二种复杂数据类型是数组。JSON 数组采用的就是 JavaScript 中的数组字面量形式。例如，下面是 JavaScript 中的数组字面量：

```
var values = [25, "hi", true];
```

在 JSON 中，可以采用同样的语法表示同一个数组：

```
[25, "hi", true]
```

7. 把数组和对象结合起来，可以构成更复杂的数据集合，例如：

```
[{
      "title": "Professional JavaScript",
      "authors": ["Nicholas C. Zakas"],
      edition: 3,
      year: 2011
},
{
      "title": "Professional JavaScript",
      "authors": ["Nicholas C. Zakas"],
```

```
      edition: 2,
      year: 2009
}
```

8. JSON 数据结构可以解析为 JavaScript 对象。就以本节第 7 点的 JSON 数据结构为例，在解析为 JavaScript 对象后，只需要下面一行简单的代码就可以取得第二本书的书名：

```
books[1].title
```

9. ECMAScript 5 定义了全局对象 JSON。JSON 对象有两个方法：stringify()和 parse()。

- stringify()：用于把 JavaScript 对象序列化为 JSON 字符串。

```
var book = {
      title: "Professional JavaScript",
      authors: ["Nicholas C. Zakas"],
      edition: 3,
      year: 2011
};
var jsonText = JSON.stringify(book);
```

- parse()：把 JSON 字符串解析为原生 JavaScript 值。

10. 有时候，JSON.stringify()还是不能满足对某些对象进行自定义序列化的需求。在这些情况下，可以给对象定义 toJSON()方法，返回其自身的 JSON 数据格式：

```
var book = {
      "title": "Professional JavaScript",
      "authors": ["Nicholas C. Zakas"],
      edition: 3,
      year: 2011,
      toJSON: function() {
            return this.title;
      }
};
var jsonText = JSON.stringify(book);
```

注意：这里的 toJSON 那个冒号应该是下面这条语句的另一种表示：

```
toJSON = function() {
            return this.title;
      }
```

因为每个函数实际上都是一个 Function 对象。

第 21 章 Ajax 与 Comet

1. Ajax，是对 Asynchronous JavaScript + XML 的简写。这一技术能够向服务器请求额外的数据而无须卸载页面，会带来更好的用户体验。

2. Ajax 技术的核心是 XMLHttpRequest 对象，简称 XHR。

3. IE7+、Firefox、Opera、Chrome 和 Safari 都支持原生的 XHR 对象，在这些浏览器中创建 XHR 对象要像下面这样使用 XMLHttpRequest 构造函数。

```
var xhr = new XMLHttpRequest();
```

4. 在使用 XHR 对象时，要调用的第一个方法是 open()，该函数打开并初始化一个请求。

```
xhr.open("get", "example.php", false);
```

有关这行代码，需要说明两点：一是 URL 相对于执行代码的当前页面（当然也可以使用绝对路径）；二是调用 open()方法并不会真正发送请求，而只是启动一个请求以备发送。

5. send()用于发送请求：

```
xhr.open("get", "example.txt", false);
xhr.send(null);
```

6. XHR 请求是同步的，JavaScript 代码会等到服务器响应之后再继续执行。在收到响应后，响应的数据会自动填充 XHR 对象的属性，相关的属性简介如下。

- responseText：作为响应主体被返回的文本。
- responseXML：如果响应的内容类型是"text/xml"或"application/xml"，这个属性中将保存包含着响应数据的 XML DOM 文档。
- status：响应的 HTTP 状态。
- statusText：HTTP 状态的说明。

7. 在接收到响应后，第一步是检查 status 属性，以确定响应已经成功返回。

```
xhr.open("get", "example.txt", false);
xhr.send(null);
if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
      alert(xhr.responseText);
} else {
      alert("Request was unsuccessful: " + xhr.status);
}
```

8. XHR 对象的 readyState 属性，该属性表示请求/响应过程的当前活动阶段。这个属性可取的值如下。

- 0：未初始化。尚未调用 open()方法。
- 1：启动。已经调用 open()方法，但尚未调用 send()方法。
- 2：发送。已经调用 send()方法，但尚未接收到响应。
- 3：接收。已经接收到部分响应数据。
- 4：完成。已经接收到全部响应数据，而且已经可以在客户端使用了。

只要 readyState 属性的值由一个值变成另一个值，都会触发一次 readystatechange 事件。

```
var xhr = createXHR();      #createXHR是个自定义的函数，创建XHR对象
xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
                  alert(xhr.responseText);
            } else {
                  alert("Request was unsuccessful: " + xhr.status);
            }
      }
};
xhr.open("get", "example.txt", true);
xhr.send(null);
```

与其他事件处理程序不同，这里没有向onreadystatechange事件处理程序中传递event对象；必须通过XHR对象本身来确定下一步该怎么做。

在接收到响应之前还可以调用abort()方法来取消异步请求，如下所示：

```
xhr.abort();
```

9. 使用setRequestHeader()方法可以设置自定义的请求头部信息。要成功发送请求头部信息，必须在调用open()方法之后且调用send()方法之前调用setRequestHeader()。

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
if (xhr.readyState == 4) {
      if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
            alert(xhr.responseText);
      } else {
            alert("Request was unsuccessful: " + xhr.status);
      }
}
xhr.open("get", "example.php", true);
xhr.setRequestHeader("MyHeader", "MyValue");
xhr.send(null);
```

服务器在接收到这种自定义的头部信息之后，可以执行相应的后续操作。我们建议读者使用自定义的头部字段名称，不要使用浏览器正常发送的字段名称，否则有可能会影响服务器的响应。

10. 调用XHR对象的getResponseHeader()方法并传入头部字段名称，可以取得相应的响应头部信息。而调用getAllResponseHeaders()方法则可以取得一个包含所有头部信息的长字符串。来看下面的例子：

```
var myHeader = xhr.getResponseHeader("MyHeader");
var allHeaders = xhr.getAllResponseHeaders();
```

11. 对XHR而言，位于传入open()方法的URL末尾的查询字符串必须经过正确的编码才行。查询字符串中每个参数的名称和值都必须使用encodeURIComponent()进行编码，

然后才能放到URL的末尾；而且所有名-值对儿都必须由和号（&）分隔，如下面的例子所示：

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

下面这个函数可以辅助向现有URL的末尾添加查询字符串参数：

```
function addURLParam(url, name, value) {
      url += (url.indexOf("?") == -1 ? "?": "&");
      url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
      return url;
}
```

12. POST请求通常用于向服务器发送应该被保存的数据。在open()方法第一个参数的位置传入"post"，就可以初始化一个POST请求，如下面的例子所示：

```
xhr.open("post", "example.php", true);
```

13. FormData为序列化表单以及创建与表单格式相同的数据（用于通过XHR传输）提供了便利。下面的代码创建了一个FormData对象，并向其中添加了一些数据：

```
var data = new FormData();
data.append("name", "Nicholas");
```

14. 而通过向FormData构造函数中传入表单元素，也可以用表单元素的数据预先向其中填入键值对儿：

```
var data = new FormData(document.forms[0]);
```

15. 创建了FormData的实例后，可以将它直接传给XHR的send()方法，如下所示：

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
                  alert(xhr.responseText);
            } else {
                  alert("Request was unsuccessful: " + xhr.status);
            }
      }
};
xhr.open("post", "postexample.php", true);
var form = document.getElementById("user-info");
xhr.send(new FormData(form));
```

16. XHR对象有一个timeout属性，表示请求在等待响应多少毫秒之后就终止。在给timeout设置一个数值后，如果在规定的时间内浏览器还没有接收到响应，那么就会触发timeout事件，进而会调用ontimeout事件处理程序。

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
            try {
                  if ((xhr.status >= 200 && xhr.status < 300) || xhr.status ==
304) {
                        alert(xhr.responseText);
                  } else {
                        alert("Request was unsuccessful: " + xhr.status);
                  }
            } catch(ex) {
                  //假设由 ontimeout 事件处理程序处理
            }
      }
};
xhr.open("get", "timeout.php", true);
xhr.timeout = 1000; //将超时设置为 1 秒钟（仅适用于 IE8+）
xhr.ontimeout = function() {
      alert("Request did not return in a second.");
};
xhr.send(null);
```

17. overrideMimeType()方法用于重写 XHR 响应的 MIME 类型。

```
var xhr = createXHR();
xhr.open("get", "text.php", true);
xhr.overrideMimeType("text/xml");
xhr.send(null);
```

18. Progress Events 规范是 W3C 的一个工作草案，定义了与客户端服务器通信有关的事件。有以下 6 个进度事件。

- loadstart：在接收到响应数据的第一个字节时触发。
- progress：在接收响应期间持续不断地触发。
- error：在请求发生错误时触发。
- abort：在因为调用 abort()方法而终止连接时触发。
- load：在接收到完整的响应数据时触发。
- loadend：在通信完成或者触发 error、abort 或 load 事件后触发。

19. 默认情况下，XHR 对象只能访问与包含它的页面位于同一个域中的资源。

20. CORS，即跨源资源共享，定义了在必须访问跨源资源时，浏览器与服务器应该如何沟通。CORS 背后的基本思想，就是使用自定义的 HTTP 头部让浏览器与服务器进行沟通，从而决定请求或响应是应该成功，还是应该失败。

比如一个简单的使用 GET 或 POST 发送的请求，它没有自定义的头部，而主体内容是 text/plain。在发送该请求时，需要给它附加一个额外的 Origin 头部，其中包含请求页面的源信息（协议、域名和端口），以便服务器根据这个头部信息来决定是否给予响应。下面是 Origin 头部的一个示例：

```
Origin: http://www.nczonline.net
```

如果服务器认为这个请求可以接受，就在 Access-Control-Allow-Origin 头部中回发相同的源信息（如果是公共资源，可以回发"*"）。例如：

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

如果没有这个头部，或者有这个头部但源信息不匹配，浏览器就会驳回请求。正常情况下，浏览器会处理请求。注意，请求和响应都不包含 cookie 信息。

21. Firefox 3.5+、Safari 4+、Chrome、iOS 版 Safari 和 Android 平台中的 WebKit 都通过 XMLHttpRequest 对象实现了对 CORS 的原生支持。在尝试打开不同来源的资源时，无需额外编写代码就可以触发这个行为。要请求位于另一个域中的资源，使用标准的 XHR 对象并在 open()方法中传入绝对 URL 即可，例如：

```
var xhr = createXHR();
xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304) {
                  alert(xhr.responseText);
            } else {
                  alert("Request was unsuccessful: " + xhr.status);
            }
      }
};
xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

22. 通过跨域 XHR 对象可以访问 status 和 statusText 属性，而且还支持同步请求。跨域 XHR 对象也有一些限制，但为了安全这些限制是必需的。以下就是这些限制。

- 不能使用 setRequestHeader()设置自定义头部。
- 不能发送和接收 cookie。
- 调用 getAllResponseHeaders()方法总会返回空字符串。

23. 默认情况下，跨源请求不提供凭据（cookie、HTTP 认证及客户端 SSL 证明等）。通过将 withCredentials 属性设置为 true，可以指定某个请求应该发送凭据。如果服务器接受带凭据的请求，会用下面的 HTTP 头部来响应。

```
Access-Control-Allow-Credentials: true
```

如果发送的是带凭据的请求，但服务器的响应中没有包含这个头部，那么浏览器就不会把响应交给 JavaScript

24. 其他跨域技术：

(1) 图像 Ping 是与服务器进行简单、单向的跨域通信的一种方式。请求的数据是通过查

询字符串形式发送的，而响应可以是任意内容，但通常是像素图或 204 响应：

```
var img = new Image();
img.onload = img.onerror = function(){
    alert("Done!");
};
img.src = "http://www.example.com/test?name=Nicholas";
```

这里创建了一个 Image 的实例，然后将 onload 和 onerror 事件处理程序指定为同一个函数。这样无论是什么响应，只要请求完成，就能得到通知。请求从设置 src 属性那一刻开始，而这个例子在请求中发送了一个 name 参数。

图像 Ping 最常用于跟踪用户点击页面或动态广告曝光次数。图像 Ping 有两个主要的缺点，一是只能发送 GET 请求，二是无法访问服务器的响应文本。因此，图像 Ping 只能用于浏览器与服务器间的单向通信。

(2) JSONP 是 JSON with padding（填充式 JSON 或参数式 JSON）的简写，JSONP 由两部分组成：回调函数和数据。回调函数是当响应到来时应该在页面中调用的函数。回调函数的名字一般是在请求中指定的。而数据就是传入回调函数中的 JSON 数据。下面是一个典型的 JSONP 请求。

```
http://freegeoip.net/json/?callback=handleResponse
```

handleResponse 是需要定义的回调函数。该技术实际上就是从其他域中加载代码执行，加载完成后执行 handleResponse()函数：

```
function handleResponse(response) {
      alert("You're at IP address " + response.ip + ", which is in " +
response.city + ", " + response.region_name);
}
var script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

(3) Ajax 是一种从页面向服务器请求数据的技术，而 Comet 则是一种服务器向页面推送数据的技术。Comet 能够让信息近乎实时地被推送到页面上，非常适合处理体育比赛的分数和股票报价。

有两种实现 Comet 的方式：长轮询和流。

(4) SSE，即服务器发送事件。SSE API 用于创建到服务器的单向连接，服务器通过这个连接可以发送任意数量的数据。服务器响应的 MIME 类型必须是 text/event-stream，而且是浏览器中的 JavaScript API 能解析格式输出。SSE 支持短轮询、长轮询和 HTTP 流，而且能在断开连接时自动确定何时重新连接。

(5) Web Sockets，也就是将协议从 HTTP 协议交换为 WebSocket 协议。使用标准的 HTTP 服务器无法实现 Web Sockets，只有支持这种协议的专门服务器才能正常工作。

25. SSE 的 JavaScript API 与其他传递消息的 JavaScript API 很相似。要预订新的事件流，首先要创建一个新的 EventSource 对象，并传进一个入口点：

```
var source = new EventSource("myevents.php");
```

26. EventSource 的实例有一个 readyState 属性，值为 0 表示正连接到服务器，值为 1 表示打开了连接，值为 2 表示关闭了连接。另外，还有以下三个事件。

- open：在建立连接时触发。
- message：在从服务器接收到新事件时触发。
- error：在无法建立连接时触发。

```
source.onmessage = function(event) {
      var data = event.data;
      //处理数据
};
```

如果想强制立即断开连接并且不再重新连接，可以调用 close()方法：

```
source.close();
```

27. 使用自定义协议而非 HTTP 协议的好处是，能够在客户端和服务器之间发送非常少量的数据，而不必担心 HTTP 那样字节级的开销。由于传递的数据包很小，因此 Web Sockets 非常适合移动应用。

28. 要创建 Web Socket，先实例一个 WebSocket 对象并传入要连接的 URL：

```
var socket = new WebSocket("ws://www.example.com/server.php");
```

注意，必须给 WebSocket 构造函数传入绝对 URL。

29. 实例化了 WebSocket 对象后，浏览器就会马上尝试创建连接。与 XHR 类似，WebSocket 也有一个表示当前状态的 readyState 属性。不过，这个属性的值与 XHR 并不相同，而是如下所示。

- WebSocket.OPENING (0)：正在建立连接。
- WebSocket.OPEN (1)：已经建立连接。
- WebSocket.CLOSING (2)：正在关闭连接。
- WebSocket.CLOSE (3)：已经关闭连接。

WebSocket 没有 readystatechange 事件；不过，它有其他事件，对应着不同的状态。readyState 的值永远从 0 开始。

30. 要关闭 Web Socket 连接，可以在任何时候调用 close()方法。

```
socket.close();
```

31. Web Socket 打开之后，就可以通过连接发送和接收数据。要向服务器发送数据，使用 send()方法并传入任意字符串，例如：

```
var socket = new WebSocket("ws://www.example.com/server.php");
socket.send("Hello world!");
```

32. Web Sockets 只能通过连接发送纯文本数据，所以对于复杂的数据结构，在通过连接

发送之前，必须进行序列化。下面的例子展示了先将数据序列化为一个 JSON 字符串，然后再发送到服务器：

```
var message = {
      time: new Date(),
      text: "Hello world!",
      clientId: "asdfp8734rew"
};
socket.send(JSON.stringify(message));
```

33. WebSocket 对象还有其他三个事件，在连接生命周期的不同阶段触发。

- open：在成功建立连接时触发。
- error：在发生错误时触发，连接不能持续。
- close：在连接关闭时触发。

34. 可以通过 XHR 访问的任何 URL 也可以通过浏览器或服务器来访问。下面的 URL 就是一个例子。

```
/getuserinfo.php?id=23
```

如果是向这个 URL 发送请求，可以想象结果会返回 ID 为 23 的用户的某些数据。谁也无法保证别人不会将这个 URL 的用户 ID 修改为 24、56 或其他值。因此，getuserinfo.php 文件必须知道请求者是否真的有权限访问要请求的数据；否则，你的服务器就会门户大开，任何人的数据都可能被泄漏出去。

35. 为确保通过 XHR 访问的 URL 安全，通行的做法就是验证发送请求者是否有权限访问相应的资源。有下列几种方式可供选择。

- 要求以 SSL 连接来访问可以通过 XHR 请求的资源。
- 要求每一次请求都要附带经过相应算法计算得到的验证码。

36. XHR 对象也提供了一些安全机制，虽然表面上看可以保证安全，但实际上却相当不可靠。实际上，前面介绍的 open()方法还能再接收两个参数：要随请求一起发送的用户名和密码。带有这两个参数的请求可以通过 SSL 发送给服务器上的页面，如下面的例子所示。

```
xhr.open("get", "example.php", true, "username", "password"); //不要这样做！！
```

即便可以考虑这种安全机制，但还是尽量不要这样做。把用户名和密码保存在 JavaScript 代码中本身就是极为不安全的。任何人，只要他会使用 JavaScript 调试器，就可以通过查看相应的变量发现纯文本形式的用户名和密码。

第 22 章 高级技巧

1. Object.preventExtensions()方法让一个对象变的不可扩展，也就是永远不能再添加新的属性。

```
var person = { name: "Nicholas" };
Object.preventExtensions(person);
```

```
person.age = 29;
alert(person.age); //undefined
```

2. Object.istExtensible()方法用于测试一个对象是否可以扩展。

```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person)); //true
Object.preventExtensions(person);
alert(Object.isExtensible(person)); //false
```

3. 密封对象不可扩展，不能删除属性和方法，但属性值是可以修改的。
4. Object.seal()方法可以让一个对象密封，并返回被密封后的对象。

```
var person = { name: "Nicholas" };
Object.seal(person);
person.age = 29;
alert(person.age); //undefined
delete person.name;
alert(person.name); //"Nicholas"
```

5. 使用 freeze()冻结对象。冻结的对象既不可扩展，又是密封的，而且对象数据属性的[[Writable]]特性会被设置为 false，即不可写。

```
var person = { name: "Nicholas" };
Object.freeze(person);
person.age = 29;
alert(person.age); //undefined
delete person.name;
alert(person.name); //"Nicholas"
person.name = "Greg";
alert(person.name); //"Nicholas"
```

6. JavaScript 是单线程运行的，其定时器到时间后并不一定会马上执行，而是会将其添加到事件处理队列中。
7. 如果 JavaScript 运行时间达到一定的限制，浏览器会弹出一个浏览器错误的对话框，告诉用户某个脚本会用过长的时间执行，询问是允许其继续执行还是停止它。可以使用定时器来解决这个问题：

```
setTimeout(function() {
      //取出下一个条目并处理
      var item = array.shift();
      process(item);
      //若还有条目，再设置另一个定时器
      if (array.length > 0) {
            setTimeout(arguments.callee, 100);
```

```
        }
},
100);
```

基本思路是每隔一段时间运行一部分。

第 23 章 离线应用与客户端存储

1. navigator.onLine 属性用于判断设备是否能上网，这个属性值为 true 表示设备能上网，值为 false 表示设备离线。

由于存在上述兼容性问题，单独使用 navigator.onLine 属性不能确定网络是否连通。即便如此，在请求发生错误的情况下，检测这个属性仍然是管用的。以下是检测该属性状态的示例。

```
if (navigator.onLine) {
      //正常工作
} else {
      //执行离线状态时的任务
}
```

2. 除 navigator.onLine 属性之外，为了更好地确定网络是否可用，HTML5 还定义了两个事件：online 事件和 offline 事件。当网络从离线变为在线或者从在线变为离线时，分别触发这两个事件。这两个事件在 window 对象上触发。

```
EventUtil.addHandler(window, "online",
function() {
      alert("Online");
});
EventUtil.addHandler(window, "offline",
function() {
      alert("Offline");
});
运行一下
```

3. Appcache 就是从浏览器的缓存中分出来的一块缓存区。要想在这个缓存中保存数据，可以使用一个描述文件（manifest file），列出要下载和缓存的资源。下面是一个简单的描述文件示例。

- CACHE MANIFEST
- #Comment
- file.js
- file.css

在最简单的情况下，描述文件中列出的都是需要下载的资源，以备离线时使用。

4. window.applicationCache 对象是对浏览器的应用缓存的编程访问方式。其 status 属性

可用于查看缓存的当前状态，属性的值是常量，表示应用缓存的如下当前状态。

- 0：无缓存，即没有与页面相关的应用缓存。
- 1：闲置，即应用缓存未得到更新。
- 2：检查中，即正在下载描述文件并检查更新。
- 3：下载中，即应用缓存正在下载描述文件中指定的资源。
- 4：更新完成，即应用缓存已经更新了资源，而且所有资源都已下载完毕，可以通过swapCache()来使用了。
- 5：废弃，即应用缓存的描述文件已经不存在了，因此页面无法再访问应用缓存

```
var appCache = window.applicationCache;

switch (appCache.status) {
  case appCache.UNCACHED: // UNCACHED == 0
    return 'UNCACHED';
    break;
  case appCache.IDLE: // IDLE == 1
    return 'IDLE';
    break;
  case appCache.CHECKING: // CHECKING == 2
    return 'CHECKING';
    break;
  case appCache.DOWNLOADING: // DOWNLOADING == 3
    return 'DOWNLOADING';
    break;
  case appCache.UPDATEREADY:  // UPDATEREADY == 4
    return 'UPDATEREADY';
    break;
  case appCache.OBSOLETE: // OBSOLETE == 5
    return 'OBSOLETE';
    break;
  default:
    return 'UKNOWN CACHE STATUS';
    break;
};
```

5. 应用缓存还有很多相关的事件，表示其状态的改变。以下是这些事件。

- checking：在浏览器为应用缓存查找更新时触发。
- error：在检查更新或下载资源期间发生错误时触发。
- noupdate：在检查描述文件发现文件无变化时触发。
- downloading：在开始下载应用缓存资源时触发。

- progress：在文件下载应用缓存的过程中持续不断地触发。
- updateready：在页面新的应用缓存下载完毕且可以通过 swapCache()使用时触发。
- cached：在应用缓存完整可用时触发。

  一般来讲，这些事件会随着页面加载按上述顺序依次触发。

6. 要以编程方式更新缓存，请先调用 applicationCache.update()。此操作将尝试更新用户的缓存（前提是已更改清单文件）。
7. 当 applicationCache.status 处于 UPDATEREADY 状态时，调用 applicationCache.swapCache() 即可将原缓存换成新缓存。
8. cookie 在性质上是绑定在特定的域名下的。当设定了一个 cookie 后，再给创建它的域名发送请求时，都会包含这个 cookie。
9. 为了最佳的浏览器兼容性，最好将整个 cookie 长度限制在 4095B（含 4095）以内。尺寸限制影响到一个域下所有的 cookie，而并非每个 cookie 单独限制。
10. cookie 由浏览器保存的以下几块信息构成。

- 名称：一个唯一确定 cookie 的名称。cookie 名称是不区分大小写的，所以 myCookie 和 MyCookie 被认为是同一个 cookie。然而，实践中最好将 cookie 名称看作是区分大小写的，因为某些服务器会这样处理 cookie。cookie 的名称必须是经过 URL 编码的。
- 值：储存在 cookie 中的字符串值。值必须被 URL 编码。
- 域：cookie 对于哪个域是有效的。所有向该域发送的请求中都会包含这个 cookie 信息。这个值可以包含子域（subdomain，如 www.wrox.com），也可以不包含它（如.wrox.com，则对于 wrox.com 的所有子域都有效）。如果没有明确设定，那么这个域会被认作来自设置 cookie 的那个域。
- 路径：对于指定域中的那个路径，应该向服务器发送 cookie。例如，你可以指定 cookie 只有从 http://www.wrox.com/books/ 中才能访问，那么 http://www.wrox.com 的页面就不会发送 cookie 信息，即使请求都是来自同一个域的。
- 失效时间：表示 cookie 何时应该被删除的时间戳（也就是，何时应该停止向服务器发送这个 cookie）。默认情况下，浏览器会话结束时即将所有 cookie 删除；不过也可以自己设置删除时间。这个值是个 GMT 格式的日期（Wdy, DD-Mon-YYYY HH:MM:SS GMT），用于指定应该删除 cookie 的准确时间。因此，cookie 可在浏览器关闭后依然保存在用户的机器上。如果你设置的失效日期是个以前的时间，则 cookie 会被立刻删除。
- 安全标志：指定后，cookie 只有在使用 SSL 连接的时候才发送到服务器。例如，cookie 信息只能发送给 https://www.wrox.com，而 http://www.wrox.com 的请求则不能发送 cookie。

11. 每一段信息都作为 Set-Cookie 头的一部分，使用分号加空格分隔每一段，如下例所示。

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; expires=Mon, 22-Jan-07 07:10:24 GMT; domain=.wrox.com
```

```
Other-header: other-header-value
```

12. cookie 设置了 secure 标志，表示该 cookie 只能通过 SSL 连接才能传输：

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value; domain=.wrox.com; path=/; secure
Other-header: other-header-value
```

13. document. cookie 属性的独特之处在于它会因为使用它的方式不同而表现出不同的行为。

- 当用来获取属性值时，document.cookie 返回当前页面可用的所有 cookie 的字符串，一系列由分号隔开的名值对儿。
- 当用于设置值的时候，document.cookie 属性可以设置为一个新的 cookie 字符串。这个 cookie 字符串会被解释并添加到现有的 cookie 集合中。设置 document.cookie 并不会覆盖 cookie，除非设置的 cookie 的名称已经存在。设置 cookie 的格式如下，和 Set-Cookie 头中使用的格式一样。

```
name=value; expires=expiration_time; path=domain_path; domain=domain_name;
secure
```

14. cookie 的所有名字和值都是经过 URL 编码的，所以必须使用 decodeURIComponent() 来解码。

15. 要给被创建的 cookie 指定额外的信息，只要将参数追加到该字符串，和 Set-Cookie 头中的格式一样，如下所示。

```
document.cookie = encodeURIComponent("name") + "="
+encodeURIComponent("Nicholas") + "; domain=.wrox.com; path=/";
```

16. 为了绕开浏览器的单域名下的 cookie 数限制，一些开发人员使用了一种称为子 cookie（subcookie）的概念。子 cookie 是存放在单个 cookie 中的更小段的数据。也就是使用 cookie 值来存储多个名称值对儿。子 cookie 最常见的的格式如下所示。

```
name=name1=value1&name2=value2&name3=value3&name4=value4&name5=value5
```

17. Web Storage 的目的是克服由 cookie 带来的一些限制，当数据需要被严格控制在客户端上时，无须持续地将数据发回服务器。Web Storage 的两个主要目标是：

- 提供一种在 cookie 之外存储会话数据的途径；
- 提供一种存储大量可以跨会话存在的数据的机制。

18. Storage 类型提供最大的存储空间（因浏览器而异）来存储名值对儿。Storage 的实例与其他对象类似，有如下方法。

- clear()： 删除所有值；Firefox 中没有实现 。

- getItem(name)：根据指定的名字 name 获取对应的值。
- key(index)：获得 index 位置处的值的名字。
- removeItem(name)：删除由 name 指定的名值对儿。
- setItem(name, value)：为指定的 name 设置一个对应的值。

19. sessionStorage 对象存储特定于某个会话的数据，也就是该数据只保持到浏览器关闭。
20. 存储在 sessionStorage 中的数据只能由最初给对象存储数据的页面访问到，所以对多页面应用有限制。由于 sessionStorage 对象其实是 Storage 的一个实例，所以可以使用 setItem()或者直接设置新的属性来存储数据。下面是这两种方法的例子。

```
//使用方法存储数据
sessionStorage.setItem("name", "Nicholas");
//使用属性存储数据
sessionStorage.book = "Professional JavaScript";
```

21. localStorage 对象在修订过的 HTML 5 规范中作为持久保存客户端数据的方案取代了 globalStorage。与 globalStorage 不同，不能给 localStorage 指定任何访问规则；规则事先就设定好了。

要访问同一个 localStorage 对象，页面必须来自同一个域名（子域名无效），使用同一种协议，在同一个端口上。这相当于 globalStorage[location.host]。

由于 localStorage 是 Storage 的实例，所以可以像使用 sessionStorage 一样来使用它。

```
//使用方法存储数据
localStorage.setItem("name", "Nicholas");
//使用属性存储数据
localStorage.book = "Professional JavaScript";
//使用方法读取数据
var name = localStorage.getItem("name");
//使用属性读取数据
var book = localStorage.book;
```

22. 存储在 localStorage 中的数据和存储在 globalStorage 中的数据一样，都遵循相同的规则：数据保留到通过 JavaScript 删除或者是用户清除浏览器缓存。
23. 对 Storage 对象进行任何修改，都会在文档上触发 storage 事件。当通过属性或 setItem()方法保存数据，使用 delete 操作符或 removeItem()删除数据，或者调用 clear()方法时，都会发生该事件。

这个事件的 event 对象有以下属性。

- domain：发生变化的存储空间的域名。
- key：设置或者删除的键名。
- newValue：如果是设置值，则是新值；如果是删除键，则是 null。
- oldValue：键被更改之前的值。

24. Indexed Database API，或者简称为 IndexedDB，是在浏览器中保存结构化数据的一种数据库。

25. IndexedDB 设计的操作完全是异步进行的。

26. IndexedDB 最大的特色是使用对象保存数据，而不是使用表来保存数据。一个 IndexedDB 数据库，就是一组位于相同命名空间下的对象的集合。

27. indexDB.open()用于打开数据库。

```
var request, database;
request = indexedDB.open("admin");
request.onerror = function(event) {
      alert("Something bad happened while trying to open: " +
event.target.errorCode);
};
request.onsuccess = function(event) {
      database = event.target.result;
};
```

28. 默认情况下，IndexedDB 数据库是没有版本号的，最好一开始就为数据库指定一个版本号。setVersion()方法用于设置版本号：

```
if (database.version != "1.0") {
      request = database.setVersion("1.0");
      request.onerror = function(event) {
            alert("Something bad happened while trying to set version: " +
event.target.errorCode);
      };
      request.onsuccess = function(event) {
            alert("Database initialization complete. Database name: " +
database.name + ", Version: " + database.version);
      };
} else {
      alert("Database already initialized. Database name: " + database.name + ",
Version: " + database.version);
```

29. 在创建对象存储空间时，必须指定这么一个键。使用 createObjectStore()方法创建对象存储空间。以下是就是为保存用户记录使用 username 作为键创建对象存储空间的示例。

```
var store = db.createObjectStore("users", { keyPath: "username" });
```

30. 使用 add()或 put()方法来向对象存储空间添加数据。如果空间中已经包含相同的键值，add()会返回错误，而 put()则会重写原有对象。简单地说，可以把 add()想象成插入新值，把 put()想象成更新原有的值。

```
//users 中保存着一批用户对象
```

```
var i = 0,
len = users.length;
while (i < len) {
      store.add(users[i++]);
}
```

31. 在数据库对象上调用 transaction()方法可以创建事务。任何时候，只要想读取或修改数据，都要通过事务来组织所有操作。在最简单的情况下，可以像下面这样创建事务：

```
var transaction = db.transaction();
```

32. 取得了事务的索引后，使用 objectStore()方法并传入存储空间的名称，就可以访问特定的存储空间。然后，可以像以前一样使用 add()和 put()方法，使用 get()可以取得值，使用 delete()可以删除对象，而使用 clear()则可以删除所有对象。

33. 游标就是一指向结果集的指针。与传统数据库查询不同，游标并不提前收集结果。游标指针会先指向结果中的第一项，在接到查找下一项的指令时，才会指向下一项。

34. 在对象存储空间上调用 openCursor()方法可以创建游标。与 IndexedDB 中的其他操作一样，openCursor()方法返回的是一个请求对象，因此必须为该对象指定 onsuccess 和 onerror 事件处理程序。

```
var store = db.transaction("users").objectStore("users"),
request = store.openCursor();
request.onsuccess = function(event) {
      //处理成功
};
request.onerror = function(event) {
      //处理失败
};
```

(1) 要检索某一个结果的信息，可以像下面这样：

```
request.onsuccess = function(event) {
      var cursor = event.target.result;
      if (cursor) { //必须要检查
            console.log("Key: " + cursor.key + ", Value: " +
JSON.stringify(cursor.value));
      }
};
```

(2) 使用游标可以更新个别的记录。调用 update()方法可以用指定的对象更新当前游标的 value。与其他操作一样，调用 update()方法也会创建一个新请求，因此如果你想知道结果，就要为它指定 onsuccess 和 onerror 事件处理程序。

(3) 调用 delete()方法，就会删除相应的记录。与 update()一样，调用 delete()也返回一个请求。

(4) 默认情况下，每个游标只发起一次请求。要想发起另一次请求，必须调用下面的一个

方法。

- continue(key)：移动到结果集中的下一项。参数 key 是可选的，不指定这个参数，游标移动到下一项；指定这个参数，游标会移动到指定键的位置。
- advance(count)：向前移动 count 指定的项数。

35. 使用游标总让人觉得不那么理想，因为通过游标查找数据的方式太有限了。键范围（key range）为使用游标增添了一些灵活性。键范围由 IDBKeyRange 的实例表示。

36. 对于某些数据，可能需要为一个对象存储空间指定多个键。比如，若要通过用户 ID 和用户名两种方式来保存用户资料，就需要通过这两个键来存取记录。为此，可以考虑将用户 ID 作为主键，然后为用户名创建索引。

(1) createIndex()方法创建索引：

```
var store = db.transaction("users").objectStore("users"),
index = store.createIndex("username", "username", { unique: false});
```

(2) 在索引上调用 openCursor()方法也可以创建新的游标，除了将来会把索引键而非主键保存在 event.result.key 属性中之外，这个游标与在对象存储空间上调用 openCursor()返回的游标完全一样。

(3) openKeyCursor()方法用于在索引上也能创建一个特殊的只返回每条记录主键的游标。

```
var store = db.transaction("users").objectStore("users"),
index = store.index("username"),
request = index.openKeyCursor();
request.onsuccess = function(event) {
      //处理成功
      // event.result.key 中保存索引键，而 event.result.value 中保存主键
};
```

(4) 使用 get()方法能够从索引中取得一个对象，只要传入相应的索引键即可；当然，这个方法也将返回一个请求。

```
var store = db.transaction("users").objectStore("users"),
index = store.index("username"),
request = index.get("007");
request.onsuccess = function(event) {
      //处理成功
};
request.onerror = function(event) {
      //处理失败
};
```

(5) 要根据给定的索引键取得主键，可以使用 getKey()方法。这个方法也会创建一个新的请求，但 event.result.value 等于主键的值，而不是包含整个对象。

37. 并发：

(1) 虽然网页中的 IndexedDB 提供的是异步 API，但仍然存在并发操作的问题。如果浏览器的两个不同的标签页打开了同一个页面，那么一个页面试图更新另一个页面尚未准备就绪的数据库的问题就有可能发生。

刚打开数据库时，要记着指定 onversionchange 事件处理程序。当同一个来源的另一个标签页调用 setVersion()时，就会执行这个回调函数。处理这个事件的最佳方式是立即关闭数据库，从而保证版本更新顺利完成。例如：

```
var request, database;
request = indexedDB.open("admin");
request.onsuccess = function(event) {
      database = event.target.result;
      database.onversionchange = function() {
            database.close();
      };
};
```

(2) 调用 setVersion()时，指定请求的 onblocked 事件处理程序也很重要。在你想要更新数据库的版本但另一个标签页已经打开数据库的情况下，就会触发这个事件处理程序。此时，最好先通知用户关闭其他标签页，然后再重新调用 setVersion()。例如：

```
ar request = database.setVersion("2.0");
request.onblocked = function() {
      alert("Please close all other tabs and try again.");
};
request.onsuccess = function() {
      //处理成功，继续
};
```

第 24 章 最佳实践

1. 一般而言，有如下一些地方需要进行注释。

- 函数和方法——每个函数或方法都应该包含一个注释，描述其目的和用于完成任务所可能使用的算法。陈述事先的假设也非常重要，如参数代表什么，函数是否有返回值（因为这不能从函数定义中推断出来）。
- 大段代码——用于完成单个任务的多行代码应该在前面放一个描述任务的注释。
- 复杂的算法——如果使用了一种独特的方式解决某个问题，则要在注释中解释你是如何做的。这不仅仅可以帮助其他浏览你代码的人，也能在下次你自己查阅代码的时候帮助理解。
- Hack——因为存在浏览器差异，JavaScript 代码一般会包含一些 hack。不要假设其他人在看代码的时候能够理解 hack 所要应付的浏览器问题。

2. 变量或函数命名的一般规则如下所示。

- 变量名应为名词如 car 或 person。
- 函数名应该以动词开始，如 getName()。返回布尔类型值的函数一般以 is 开头，如 isEnable()。
- 变量和函数都应使用合乎逻辑的名字，不要担心长度。长度问题可以通过后处理和压缩（本章后面会讲到）来缓解。

3. 耦合过紧的代码可维护难度高，因此尽可能维护弱耦合的代码:

(1) 解耦 HTML/JavaScript：也就是在 HTML 中不要出现 HTML 语句，也不要在 HTML 文件中使用 JavaScript 语句，如下所示：

```
//将 HTML 紧密耦合到 JavaScript
function insertMessage(msg) {
      var container = document.getElementById("container");
      container.innerHTML = "<div class=\"msg\"><p class=\"post\">" + msg +
"</p>" + "<p><em>Latest message above.</em></p></div>";
}
```

(2) 解耦 CSS/JavaScript：也就是在 HTML 中不要出现 CSS 语句及进行 CSS 设置，如下所示：

```
//CSS 对 JavaScript 的紧密耦合
element.style.color = "red";
element.style.backgroundColor = "blue";
```

(3) 解耦应用逻辑 / 事件处理程序

4. 增加脚本的整体性能：

- 避免全局查找
- 避免 with 语句

5. 可以使用压缩工具减少 JavaScript 文件的大小。