1. **C++和C的主要区别**
2. 语言本身：C++面向对象，C面向过程；C以结构体struct为核心结构，C++以类class为核心结构；C以宏定义的方式部分支持多态，C++自身提供多态，模版支持编译期多态，虚函数支持运行期多态。
3. const修饰的量

C中const修饰的是常变量，不是常量，不能被修改，不是必须初始化 如const int a=10；

C++中const修饰的是常量，必须初始化

1. 内存分配：C中使用malloc和free，C++中建议使用new/delete。相对于malloc，new除了有开辟内存的功能，还有初始化的功能对于内置类型，调用构造函数对于自定义函数。delete不同于free，先进行析构，在free释放内存。
2. C++支持函数重载（C++允许拥有不同变量但具有相同函数名的函数），C不支持
3. C++中有引用
4. **C++中的引用和C中的指针**

a. C++中引用必须初始化，初始化的值必须要能取地址（例：int &a=10;//错误），但是不分配存储空间。C中指针不声明时初始化，在初始化的时候需要分配存储空间。引用相对于指针更加安全，因为指针可指向任意一块内存，可以改变所指的对。

b. 指针是一个新的变量，存储了另一个变量的地址，我们可以通过访问这个地址来修改另一个变量；引用只是一个别名，还是变量本身，对引用的任何操作就是对变量本身进行操作，以达到修改变量的目的

c. 引用的大小根据所引用的类型确定，指针的大小由编译器位数以及操作系统位数确定。引用只有一级，指针可以多级。

d. 指针传参的时候，还是值传递，指针本身的值不可以修改，需要通过解引用才能对指向的对象进行操作；引用传参的时候，传进来的就是变量本身，因此变量可以被修改

e. 指针可以指向空值。

1. **vector与map的区别/list, vector, map, set对比**

**1）List封装了链表,Vector封装了数组, list和vector得最主要的区别在于vector使用连续内存存储的，他支持[]运算符，而list是以链表形式实现的，不支持[]。Vector对于随机访问的速度很快，但是对于插入尤其是在头部插入元素速度很慢，在尾部插入速度很快。**

**List对于随机访问速度慢得多，因为可能要遍历整个链表才能做到，但是对于插入就快的多了，不需要拷贝和移动数据，只需要改变指针的指向就可以了。另外对于新添加的元素，Vector有一套算法，而List可以任意加入。  
2）Map,Set属于标准关联容器，使用了非常高效的平衡检索二叉树：红黑树，他的插入删除效率比其他序列容器高是因为不需要做内存拷贝和内存移动，而直接替换指向节点的指针即可。**

**Set和Vector的区别在于Set不包含重复的数据。**

**Set和Map的区别在于Set只含有Key，而Map有一个Key和Key所对应的Value两个元素。  
Map和Hash\_Map的区别是Hash\_Map使用了Hash算法来加快查找过程，但是需要更多的内存来存放这些Hash桶元素，因此可以算得上是采用空间来换取时间策略。**

**Vector和map的区别是vector是一种顺序容器，每个元素有对应的位置，map是一种关联容器，元素按关键字的值存储，建立起对应关系。**

1. **C++STL迭代器失效问题**

**STL迭代器**通俗一点来讲就是用来遍历容器的，并对容器进行一定的操作。定义好底层构造，任何容器都以同样的遍历方式，即使不懂底层实现也可以操作容器。

在对容器进行操作的时候，由于一些操作，使得元素保存的位置发生改变，从而使得对指向原来位置的迭代器操作时发生错误。

**失效类型**：

a.由于插入元素，使得容器元素整体“迁移”导致存放原容器元素的空间不再有效，从而使得指向原空间的迭代器失效。

b.由于删除元素使得某些元素次序发生变化使得原本指向某元素的迭代器不再指向希望指向的元素。

**失效场景：**

**Vector**：内部结构为数组，插入时出现扩容的情况/删除时，指向删除节点的迭代器会失效

解决方法：在出现迭代器失效的位置重新获取一下迭代器。

**List**：由于list底层是一个带头节点的双向链表，所以不会出现扩容，因此list迭代器失效的主要场景是删除(earse)操作。

解决方法：重新获取新的迭代器。it = li.erase(it);

**Deque**：内部结构为数组，增加任何元素都将使deque的迭代器失效。在deque的中间删除元素将使迭代器失效。在deque的头或尾删除元素时，只有指向该元素的迭代器失效。

**set**：键唯一。元素默认按升序排列。如果迭代器所指向的元素被删除，则该迭代器失效。其它任何增加、删除元素的操作都不会使迭代器失效。

**Map**：键唯一。元素默认按键的升序排列。如果迭代器所指向的元素被删除，则该迭代器失效。其它任何增加、删除元素的操作都不会使迭代器失效。

1. **虚函数**

C++中的虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”。 多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。

虚函数的实现是由两个部分组成的，虚函数指针与虚函数表。**虚函数指针**本质上说一个指向指针的指针，只有拥有虚函数的类才会拥有虚函数指针，每一个虚函数也都会对应一个虚函数指针。所以拥有虚函数的类的所有对象都会因为虚函数产生额外的开销，并且也会在一定程度上降低程序速度。每个类的实例化对象都会拥有虚函数指针并且都**排列在对象的地址首部**。而它们也都是按照一定的顺序组织起来的，从而构成了一种表状结构，称为虚函数表。

**虚函数表** 主要是一个类的虚函数的地址表，这张表被分配在有虚函数的类的实例的内存中，当父类的指针操作一个子类的时候，能指明实际调用的函数。C++的编译器保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。（虚函数表的大小在变异时期确定，不需要动态分配，故不在堆中，类似于类中的静态变量，位于只读数据段，即常量段；而虚函数位于代码段，即代码区）

**构造函数不可以是虚函数**：虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数——构造函数了

**析构函数可以是虚函数**，而且当要使用基类指针或引用调用子类时，最好将基类的析构函数声明为虚函数，否则可以存在内存泄露的问题。

举例说明：子类B继承自基类A；A \*p = new B; delete p;

1）如果类A的析构函数不是虚函数，那么delete p；将会仅仅调用A的析构函数，只释放了B对象中的A部分，而派生出的新的部分未释放掉。

2） 如果类A的析构函数是虚函数，delete p; 将会先调用B的析构函数，再调用A的析构函数，释放B对象的所有空间。

**C++默认的析构函数不是虚函数**是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数

**虚函数不能inline**：inline函数需要在编译期将函数体替换到函数调用处，而虚函数是在运行时才能确定调用的是哪个函数，因此在编译期无法确定调用的是父类的还是子类的虚函数。

**静态成员函数不可以是虚函数：**静态成员函数是以类为单位的函数，与具体对象无关，虚函数是与对象动态绑定的，因此是两个不冲突的概念。

**纯虚函数**：在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”。如virtual void funtion1()=0。纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

**必要性**：1）方便使用多态性

2）在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例

**抽象类：**称带有纯虚函数的类为抽象类，抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

性质：1）抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

2）抽象类是不能定义对象的。

**C++虚函数与虚继承实现机制**

虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题，通常可以将派生类对象的地址赋值给基类对象，实现的具体方式是，将基类指针指向继承类（继承类有基类的拷贝）中的基类对象的地址，但是多重继承可能存在一个基类的多份拷贝，这就出现了二义性。虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。虚基类指针指向一个虚基类表，虚基类表中记录着虚基类和本类的地址偏移，通过地址偏移可以找到虚基类成员，所以虚继承不需要像普通多即成一样多次拷贝，节省了存储空间。

**一个对象访问普通成员函数和虚函数哪个更快**？

访问普通成员函数更快，因为普通成员函数的地址在编译阶段就已确定，因此在访问时直接调用对应地址的函数，而虚函数在调用时，需要首先在虚函数表中寻找虚函数所在地址，因此相比普通成员函数速度要慢一些

1. **new/malloc与delete/free的区别**

int p1=(int)malloc(size); int \*p2 = new int[size]; delete []p2

**new和delete / new[ ]和delete[ ] 一定要匹配使用**

1. malloc/free需要库文件支持，而new/delete 是C++的运算符，不需要库文件支持
2. C++中new/delete能调用构造函数和析构函数，而malloc/free不能，只是单纯的申请内存空间

3）new 返回指定类型的指针，并且可以自动计算所需要大小(因为new内置了sizeof、类型转换和类型安全检查)，而malloc 需要手工计算字节数，并且在返回后强行转换为实际类型的指针。new是类型安全的，malloc不是类型安全的

4）C中的malloc分配内存失败直接返回NULL，new在分配内存失败时默认的操作是抛出一个内置的异常，标准的C++也提供了抑制抛出异常的方法，使之在内存分配出错时不再抛出异常而是直接返回NULL

5）new可以重载，可以自定义内存分配策略，甚至不做内存分配，甚至分配到非内存设备上。而malloc无能为力

**有了malloc/free为什么还要new/delete？**

对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。

**既然new/delete的功能完全覆盖了malloc/free，为什么C++不把malloc/free淘汰出局呢**

因为C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用delete释放“malloc申请的动态内存”，虽然理论上程序不会出错，但是该程序的可读性很差。所以new/delete必须配对使用，malloc/free也一样

**New/delete底层实现**

1. new/new[]：完成两件事，先底层调用malloc分了配内存，然后创建一个对象（调用构造函数）。**new ---->operator new ----> malloc**

2. delete/delete[]:也完成两件事，先调用析构函数（清理资源），然后底层调用free释放空间。**delete --->operator delete --->free**

1. **C++模板特化/偏特化**

C++中的模板分为类模板和函数模板，不知道用什么数据类型时建立一个模板。

有时候为了需要，针对特定的类型，需要对模版进行特化，即特殊处理。模板的偏特化是指需要根据模板的某些但不是全部的参数进行特化，即对部分参数做特殊处理。函数模版只能全特化不能偏特化

1. **如何禁用自定义类型的拷贝构造函数与赋值运算符？**

1）将复制构造函数和赋值操作符声明为private。

2）在想要 “禁止使用” 的特殊成员函数声明后加 “= delete”

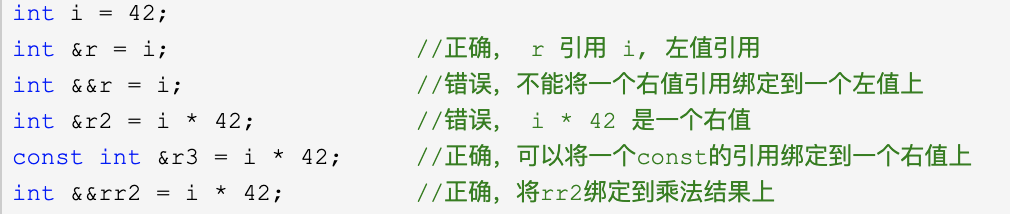
3）若也不允许友元和成员使用，只提供成员的声明而不提供定义。这样当程序中出现复制或赋值现象时，会造成链接错误

1. **右值引用与移动构造函数**

**左值引用** 为已经存在的对象创建一个新的名字。对新引用所做的访问和修改操作，都会影响它的原型。

右值引用 只能绑定右值，不能绑定左值，它使用两个&&来声明

**一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值**

****

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。右值引用指向将要被销毁的对象。因此，可以从绑定到右值引用的对象“窃取”状态，即使用右值引用的代码可以自由接管所引用的对象的资源。不能将右值直接绑定到一个左值上，但可以显式地将左值转换为对应的右值引用类型（move函数）int &&rr3 = std::move(rr1);

**移动构造函数**类似拷贝构造函数，第一个参数是该类类型的一个引用。不同于拷贝构造函数的是，这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数相同，任何额外的参数都必须有默认实参。

string (const string& str);//拷贝构造

string (string&& str) noexcept;//移动构造

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态 — 销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源。（将动态数据移动到新对象中后，应该解除与源对象的关系）这些资源的所有权已经归属于新创建的对象

**移动赋值运算符** 与移动构造函数类似，构造函数在创建或初始化对象的时候调用，而赋值运算符在更新一个对象的值时调用。

**引入移动构造函数以及移动赋值就是为了利用充分利用临时变量，减少不必要的空间的申请与释放**

1. **智能指针**

除了静态内存和栈内存外，每个程序还有一个内存池，这部分内存被称为自由空间或者堆。程序用堆来存储动态分配的对象即那些在程序运行时分配的对象，当动态对象不再使用时，我们的代码必须显式的销毁它们。在C++中，动态内存的管理是用一对运算符完成的：new和delete，new:在动态内存中为对象分配一块空间并返回一个指向该对象的指针，delete：指向一个动态独享的指针，销毁对象，并释放与之关联的内存。动态内存管理经常会出现两种问题：一种是忘记释放内存，会造成内存泄漏；一种是尚有指针引用内存的情况下就释放了它，就会产生引用非法内存的指针。**为了更加容易（更加安全）的使用动态内存，引入了智能指针的概念**。智能指针的行为类似常规指针，重要的区别是它**负责自动释放所指向的对象**

1）智能指针是通过基本类型（模板类）指针，构造类的对象，只能指针本身就是一个自定义的对象。它将普通的指针封装为一个栈对象

2）当此对象被销毁时，即调用此对象的析构函数，释放此指针。

**也就是用栈中的空间来管理堆中的内存**。

标准库提供的两种智能指针的区别在于管理底层指针的方法不同，shared\_ptr允许多个指针指向同一个对象，unique\_ptr则“独占”所指向的对象。标准库还定义了一种名为weak\_ptr的伴随类，它是一种弱引用，指向shared\_ptr所管理的对象。

**shared\_ptr**维护了一个指向control block的指针对象，来记录引用个数；

**weak\_ptr**用于避免shared\_ptr相互指向产生的环形结构，造成的内存泄漏。weak\_ptr count是弱引用个数；弱引用个数不影响shared count和对象本身，shared count为0时则直接销毁。

**如何判断weak\_ptr的对象是否失效？**

1）expired()：检查被引用的对象是否已删除。

2）lock()会返回shared指针，判断该指针是否为空。

3）use\_count()也可以得到shared引用的个数，但速度较慢。

1. Vector/map变量生命周期结束后内存没有释放，只是把vector/map清空了，但是内存没有释放，但在下次分配时会复用该块内存；如果要释放内存不止是要clear()掉，还要和一个空的map来进行swap，将内存释放。
2. **C++顶层与底层const**

**底层const是代表对象本身是一个常量（不可改变）；**

**顶层const是代表指针的值是一个常量,而指针的值(即对象的地址)的内容可以改变（指向的不可改变）**

**一个指针本身添加const限定符就是顶层const，而指针所指的对象添加const限定符就是底层const。**

1）执行对象拷贝时有限制，常量的底层const不能赋值给非常量的底层const

2）使用命名的强制类型转换函数const\_cast时，需要能够分辨底层const和顶层const，因为const\_cast只能改变运算对象的底层const

1. 静态编译 动态编译 静态库 动态库

**静态链接&动态链接**

静态链接的过程就已经把要链接的内容已经链接到了生成的可执行文件中，就算你在去把静态库删除也不会影响可执行程序的执行（缺点是浪费空间且难更新，优点是速度快）；而动态链接这个过程却没有把内容链接进去，而是在执行的过程中，再去找要链接的内容，生成的可执行文件中并没有要链接的内容，所以当你删除动态库时，可执行程序就不能运行。

**静态库&动态库**

静态连接库就是把(lib)文件中用到的函数代码直接链接进目标程序，程序运行的时候不再需要其它的库文件；动态链接就是把调用的函数所在文件模块（DLL）和调用函数在文件中的位置等信息链接进目标程序，程序运行的时候再从DLL中寻找相应函数代码，因此需要相应DLL文件的支持。

1. **内联函数/普通函数/宏**

函数的调用必须要将程序执行的顺序转移到函数所存放在内存中的某个地址，将函数的程序内容执行完后，再返回到转去执行该函数前的地方。这种转移操作要求在转去执行前要保存现场并记忆执行的地址，转回后要恢复现场，并按原来保存地址继续执行。因此，函数调用要有一定的时间和空间方面的开销，于是将影响其效率。而**宏只是在预处理的地方把代码展开，不需要额外的空间和时间方面的开**销，所以调用一个宏比调用一个函数更有效率

**内联函数**"就是将很简单的函数"内嵌"到调用他的程序代码中；内联函数的执行过程与带参数宏定义很相似，但参数的处理不同。带参数的宏定义并不对参数进行运算，而是直接替换；内联函数首先是函数，先把参数表达式进行运算求值，然后把表达式的值传递给形式参数。

**内联函数&宏**

1） 宏做的是简单的字符串替换,而函数的参数的传递,参数是有数据类型的,可以是各种各样的类型.

2） 宏的参数替换是不经计算而直接处理的,而函数调用是将实参的值传递给形参,值是计算得来的.

3） 宏在编译之前进行,即先用宏体替换宏名,然后再编译的,而函数显然是编译之后,在执行时,才调用的.因此,宏占用的是编译的时间,而函数占用的是执行时的时间.

4） 宏的参数是不占内存空间的,因为只是做字符串的替换,而函数调用时的参数传递则是具体变量之间的信息传递,形参作为函数的局部变量,显然是占用内存的.

5） 函数的调用是需要付出一定的时空开销的,因为系统在调用函数时,要保留现场,然后转入被调用函数去执行,调用完,再返回主调函数,此时再恢复现场,这些操作,显然在宏中是没有的.

**内联函数&普通函数**

1）在编译过程中，内联函数在函数的调用点，把函数代码全部展开，所以没有标准函数的栈帧的开辟和回退。（没有寻址过程）

2）内联函数只在本文件可见，编译阶段就进行了替换，所以不产生符号，所以一般在头文件中定义，这样就可以在其它文件调用。普通函数产生符号，多个文件引用头文件，会产生符号重定义的错误。

**静态函数&普通函数**

static函数和普通函数的最大的区别在于作用域方面，static函数限定在本源码文件中，不能被本源码文件以外的代码文件调用。而普通的函数，默认是extern的，也就是说，可以被其它代码文件调用该函数。同时static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝

1. **类默认函数**

构造函数，拷贝构造函数，析构函数，重载赋值运算符函数，重载取址运算符函数，重载取址运算符const函数，移动构造函数，重载移动赋值操作符函数

1. **大小端模式**

**大端（存储）模式**，是指数据的低位保存在内存的高地址中，而数据的高位保存在内存的低地址中  
**小端（存储）模式**，是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中

需要**字节对齐**的根本原因在于CPU访问数据的效率问题。假设地址不是自然对齐，则CPU如果取它的值的话需要访问多次内存，然后组合得到所要的数据，而如果变量在自然对齐位置上，则只要一次就可以取出数据

对于标准数据类型，它的地址只要是它的长度的整数倍就行了，而非标准数据类型按下面的原则对齐：

　　数组 ：按照基本数据类型对齐，第一个对齐了后面的自然也就对齐了。

　　联合 ：按其包含的长度最大的数据类型对齐。

　　结构体： 结构体中每个数据类型都要对齐。

取消字节对齐：pragma pack()

1. **如何初始化const和static数据成员？**

通常在类外申明static成员，但是static const的整型（bool，char，int，long）可以在类中声明且初始化，static const的其他类型必须在类外初始化（包括整型数组）。

**Static的作用**

static修饰局部变量->静态局部变量（存储在静态存储区，改变了位置，原来在栈，未初始化会自动初始化为0，作用域为局部作用域）

static修饰全局变量->静态全局变量（静态存储区，自动初始化为0，在声明他的文件之外是不可见的）

static修饰类成员变量->类的全局变量，被类的所有对象共享，必须在类外初始化

static修饰成员函数，这个类只存在这一份函数，所有对象共享该函数

**const的作用：**

1.限定变量为不可修改。

2.限定成员函数不可以修改任何数据成员。

3.const与指针：

const char \*p 表示 指向的内容不能改变。

char \* const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

1. **C++的内存分区**

栈区：主要存放函数参数以及局部变量，由系统自动分配释放

堆区：存放动态变量，手动释放

全局/静态区：存放全局变量、静态变量；程序结束后由系统释放

字符串常量区：字符串常量就放在这里，程序结束后由系统释放。

代码区：存放程序的二进制代码

1. **Const和define的区别**

1）编译器处理方式

define – 在预处理阶段进行替换 const – 在编译时确定其值

2）类型检查

define – 无类型，不进行类型安全检查，可能会产生意想不到的错误   
const – 有数据类型，编译时会进行类型检查

3）内存空间

define – 不分配内存，给出的是立即数，有多少次使用就进行多少次替换，在内存中会有多个拷贝，消耗内存大

const – 在静态存储区中分配空间，在程序运行过程中内存中只有一个拷贝

4）其他   
在编译时， 编译器通常不为const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。宏替换只作替换，不做计算，不做表达式求解。

1. 内存池

进行一次性的开出大块空间，当你需要构造对象的时候，拿出来构造，当你需要释放的时候，并不还给操作系统，还给内存池，然后接下来使用直接从内存池当中取出进行操作。在多次进行分配请求与释放请求的场景下，内存池能够提高效率。简单的实现成为一个自由链表的形式。

1. **jni封装的具体步骤**

1）配置java环境

2）实现一个类

3）编写add.cpp封装类Twono中的add（）

4）将该add.cpp生成动态连接库

5）重新编译

1. **装饰者模式/策略模式/观察者模式**

装饰者模式：给一个对象增加一些新的功能，并且是动态的，要求装饰对象和被装对象实现同一个接口，装饰对象持有被装对象的实例。

策略模式：定义一系列算法，并将每个算法封装起来，是他们可以相互替换，算法的变化不影响使用算法的客户。要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数

观察者模式：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系

1. **进程/线程/协程**

**进程**是**资源分配和调度的基本单位**，是程序的基本执行实体，是指令，数据及其组织形式的描述。进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

**特征** 1）动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。 2）并发性：任何进程都可以同其他进程一起并发执行 3）独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位； 4）异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进 5）结构特征：进程由程序、数据和进程控制块三部分组成。

多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进程，能得到不同的结果；但是执行过程中，程序不能发生改变。

**状态**：就绪态，运行态，阻塞态

**线程**是程序执行流的最小单元。一个标准的线程由线程ID，当前指令指针(PC），寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是**被系统独立调度和分派的基本单位**，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。

特征：1）轻型实体 2）独立调度和分派的基本单位，线程的切换非常迅速且开销小（在同一进程中） 3）可并发执行。 在一个进程中的多个线程之间，可以并发执行，甚至允许在一个进程中所有线程都能并发执行；同样，不同进程中的线程也能并发执行，充分利用和发挥了处理机与外围设备并行工作的能力。 4）共享进程资源。在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的地址空间（进程的地址空间），这意味着，线程可以访问该地址空间的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。

**协程**是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

**进程和线程的比较**

1、地址空间: 线程是进程内的一个执行单元，共享进程的地址空间;而进程有自己独立的地址空间;因此线程可以读写同样的数据结构和变量，便于线程之间的通信。相反，进程间通信（IPC）很困难且消耗更多资源。

2、资源拥有:进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源

3、进程是资源的分配和调度的一个独立单元，而线程是CPU调度的基本单元

4、二者均可并发执行.

5、进程的创建调用fork或者vfork，而线程的创建调用pthread\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

6、线程有自己的私有属性TCB，线程id，寄存器、硬件上下文，而进程也有自己的私有属性进程控制块PCB，这些私有属性是不被共享的，用来标示一个进程或一个线程的标志

**线程和协程的比较**

1极高的执行效率：因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显；

2 不需要多线程的锁机制：因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多

进程切换包括1）切换页目录使用新的地址空间2）切换内核栈及硬件上下文。

线程切换不需要进行第一步，第二步进程/线程切换都需要做

**进程切换代价**：上下文切换的开销，线程本身的开销（内存同步，线程创建和消亡，切换，调度等）

1. **进程调度算法**

服务时间：作业需要运行的时间  
完成时间 = 开始时间 + 服务时间  
等待时间 = 开始时间 - 提交时间  
周转时间 = 完成时间 - 提交时间  
带权周转时间 = 周转时间 / 服务时间  
响应比 = （等待时间 + 服务时间） / 服务时间 = 等待时间/服务时间 + 1

1. 先来先服务FCFS
2. 短作业优先
3. 高响应比优先
4. 时间片轮转
5. 多级反馈队列
6. **互斥锁，读写锁与自旋锁**

**互斥锁**：共享资源的使用是互斥的，即一个线程获得资源的使用权后就会将改资源加锁，使用完后会将其解锁，所以在使用过程中有其它线程想要获取该资源的锁，那么它就会被阻塞陷入睡眠状态，直到该资源被解锁才会别唤醒，如果被阻塞的资源不止一个，那么它们都会被唤醒，但是获得资源使用权的是第一个被唤醒的线程，其它线程又陷入沉睡

**递归锁：**同一个线程可以多次获得该资源锁，别的线程必须等待该线程释放所有次数的锁才能获得。

**读写锁：**读写锁拥有读状态加锁、写状态加锁、不加锁三种状态。只有一个线程可以占有写状态的锁，但可以多个线程同时占有读状态锁，这也是它可以实现高并发的原因。当其处于写状态锁下，任何想要尝试获得锁的线程都会被阻塞，直到写状态锁被释放；如果是处于读状态锁下，允许其它线程获得它的读状态锁，但是不允许获得它的写状态锁，当读写锁感知到有线程想要获得写状态锁时，便会阻塞其后所有想要获得读状态锁的线程。所以读写锁非常适合资源的读操作远多于写操作的情况。

**读写锁三个特征**：1）多个读者可以同时进行读2）写者必须互斥，只允许一个写者写，也不能读者写者同时进行3）写者优先于读者，一旦有写者，则后续读者必须等待，唤醒时优先考虑写者

**自旋锁**是一种特殊的互斥锁，当资源被加锁后，其它线程想要再次加锁，此时该线程不会被阻塞睡眠而是陷入循环等待状态（不能再做其它事情），循环检查资源持有者是否已经释放了资源，这样做的好处是减少了线程从睡眠到唤醒的资源消耗，但会一直占用CPU资源。适用于资源的锁被持有的时间短，而不希望在线程的唤醒上花费太多资源的情况。自旋锁的实现是为了保护一段短小的临界区操作代码，保证这个临界区的操作是原子的，从而避免并发的竞争冒险。在Linux内核中，自旋锁通常用于包含内核数据结构的操作。

1. **用户态与内核态的区别以及区分的原因**

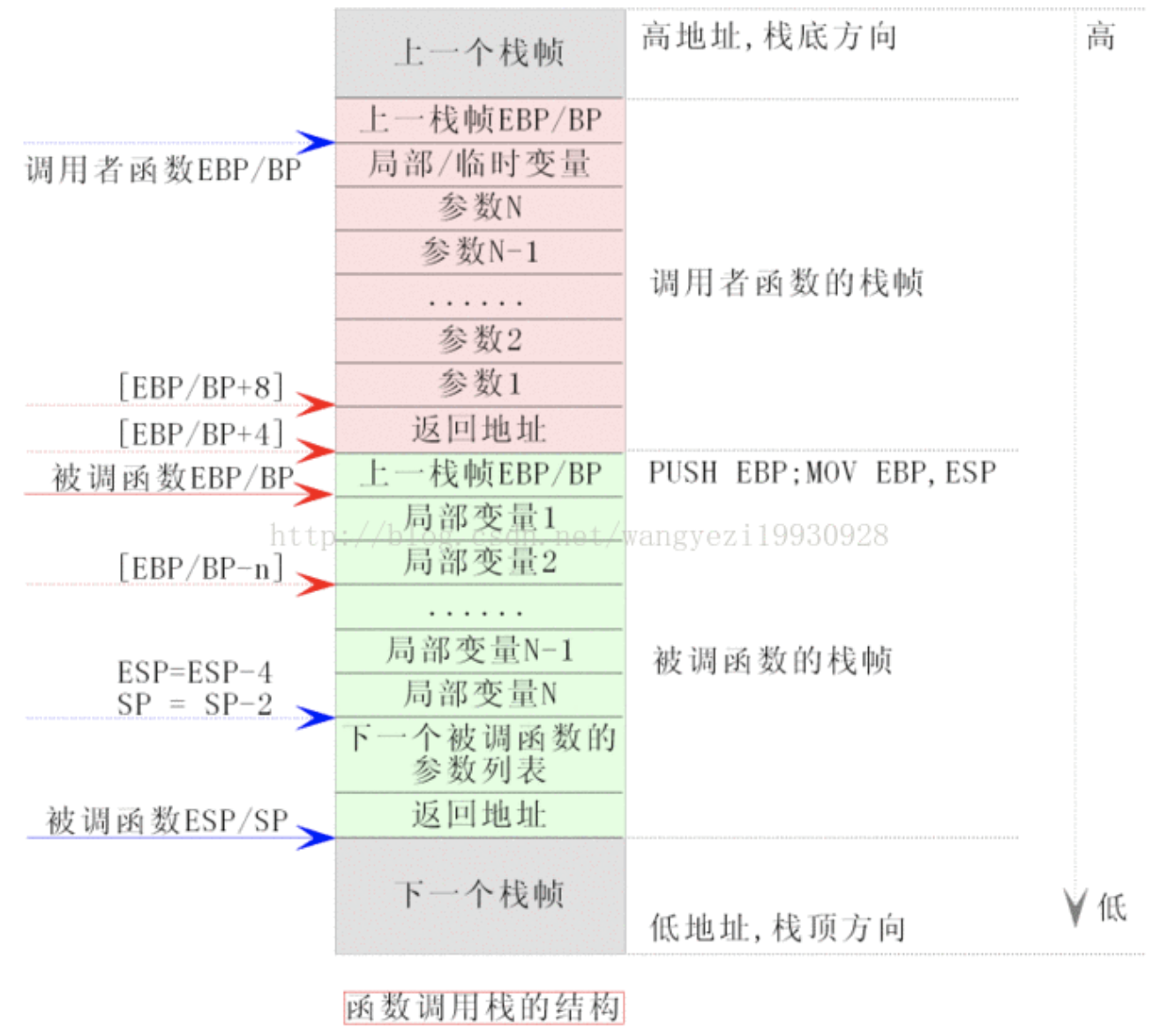
用户运行一个程序，该程序创建的进程开始时运行自己的代码，处于**用户态（特权级3）**。如果要执行文件操作、网络数据发送等操作必须通过 write、send 等系统调用，这些系统调用会调用内核的代码，进入**内核态（特权级0）**。进程会切换到 Ring0，然后进入 3G-4G 中的内核地址空间去执行内核代码来完成相应的操作。内核态的进程执行完后又会切换到 Ring3，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。这说的保护模式是指通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程地址空间中的数据。

用户态切换到内核态的3种方式：（1）系统调用（2）异常 （3）外围设备的中断

**区分原因**：1. 对于任何操作系统来说，创建一个进程是核心功能。创建进程要做很多工作，会消耗很多物理资源。比如分配物理内存，父子进程拷贝信息，拷贝设置页目录页表等等，这些工作得由特定的进程去做，所以就有了特权级别的概念。最关键的工作必须交给特权级最高的进程去执行，这样可以做到集中管理，减少有限资源的访问和使用冲突。inter x86架构的cpu一共有四个级别，0-3级，0级特权级最高，3级特权级最低。2.在CPU的所有指令中，有一些指令是非常危险的，如果错用，将导致整个系统崩溃。比如：清内存、设置时钟等。如果所有的程序都能使用这些指令，会发生很多错误。所以，CPU将指令分为特权指令和非特权指令，对于那些危险的指令，只允许操作系统及其相关模块使用，普通的应用程序只能使用那些不会造成灾难的指令。

**主要差别**：处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理机是可被抢占的 ； 而处于核心态执行中的进程，则能访问所有的内存空间和对象，且所占有的处理机是不允许被抢占的。

1. **函数调用中栈的作用**
2. 当发生函数调用时，调用函数把被调函数所需要的参数按照与被调函数形参顺序相反的顺序压入栈（即从右到左）
3. 调用函数使用call指令调用被调函数并把call指令的下一条指令的地址当成返回地址压入栈中
4. 在被调函数中，被调函数会先保存调用函数的栈底地址，然后再保存栈顶地址，即当前被调函数的栈底地址
5. 在被调函数中，从栈底的位置开始存放被调函数中的局部变量和临时变量，这些变量的地址按定义顺序依次减小，即先定义的先入栈

****

1. **操作系统内存管理碎片问题**

**外部碎片**：未被分配的内存，太多不连续的小内存无法满足较大内存的申请要求；

产生原因：频繁的分配和回收物理页导致大量的小块内存夹杂在已分配的页中间

**内部碎片**：已经分配却不能被利用的内存空间

产生原因：所有内存分配必须起始可被4，8，16整除的地址或者MMU分页机制限制

1. **页面置换算法**

最佳置换算法OPT：置换出最长时间不被使用的页面

最近最久未使用置换算法LRU

最近未使用置换算法NRU

先进先出置换算法FIFO

第二次机会算法

时钟置换算法CLOCK：将所有的页组成一个圆，圆心的指针指向下一个要被置换的页面，置换前检查使用位，如果使用位为1，将其使用位置为0，随后将顺指针旋转，检查下一个页面，直到发现某页的使用位为0，将此页置换出内存。

1. **乐观锁与悲观锁**

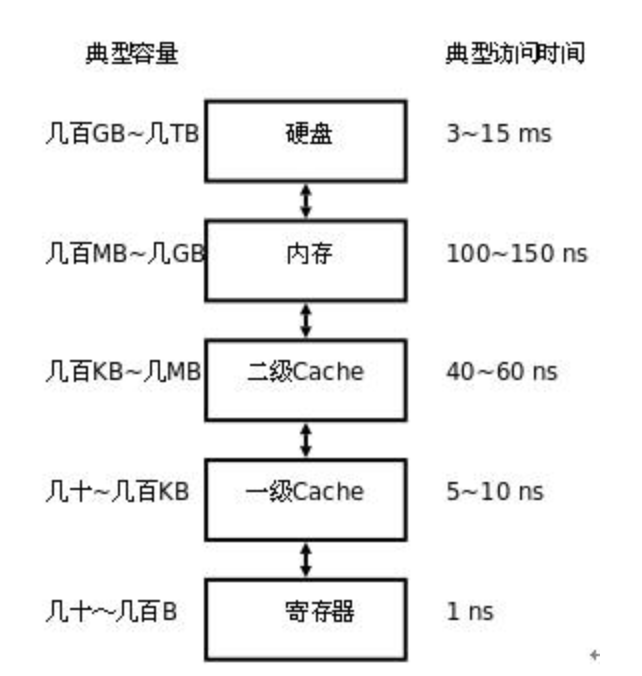
**悲观锁**：总是假设最坏的情况，每次取数据的时候都认为别人会修改，所以每次取数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。如行锁，表锁，读写锁

**乐观锁**：总是假设最好的情况，每次取数据别人都不会修改，所以不会上锁，但在更新的时候判断一下在此期间别人有没有更新数据。多用于多读的情况，可以提高吞吐量

1. **Cache**

作用：作为缓冲器缓解高低速设备之间的通信延迟问题（cpu和内存）

分为3级，速度依次递减，容量依次递增，直接映射/组相连/全相连。分级是为了既快又节约成本



1. **二叉树**

1）**二叉查找树**：左子树的所有节点的值都比右子树小

**二叉搜索树**：二叉树的各个节点记录的数值是有序排列的，它有以下三条性质：

\* 左子节点值都小于根节点值

\* 右子节点值都大于根节点值

\* 所有节点的子树也都是二叉搜索树

当二叉搜索树的每个节点都有两个子节点时，树才是最优的，极端情况会退化成链表

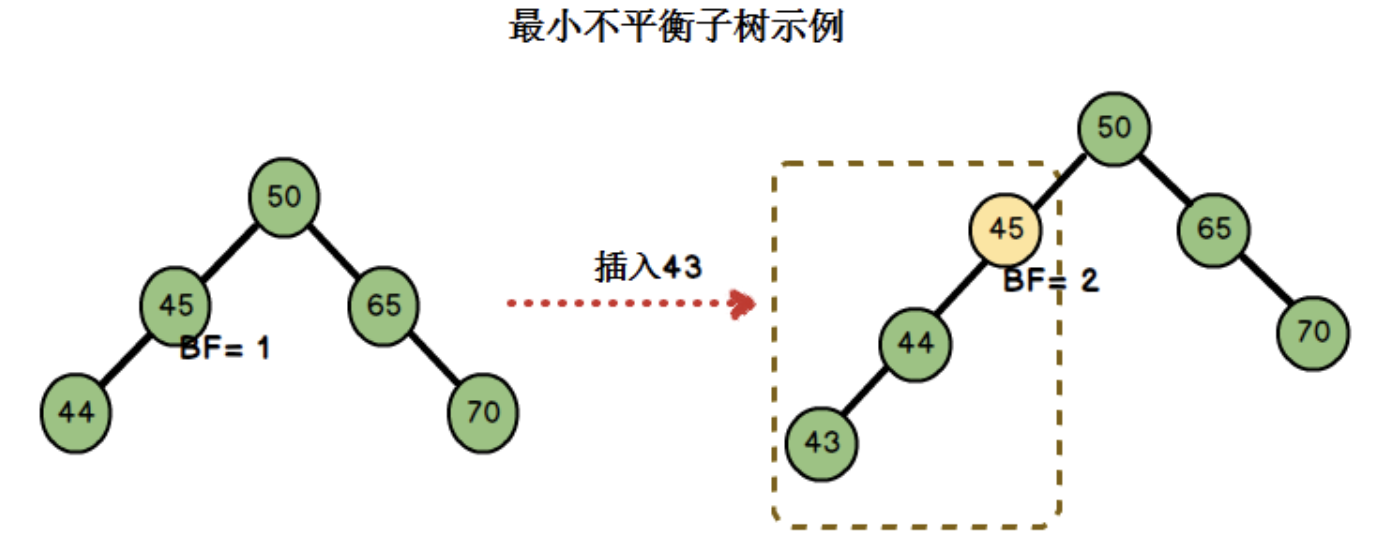
**平衡树：AVL/红黑树/SBT/Treap/伸展树**

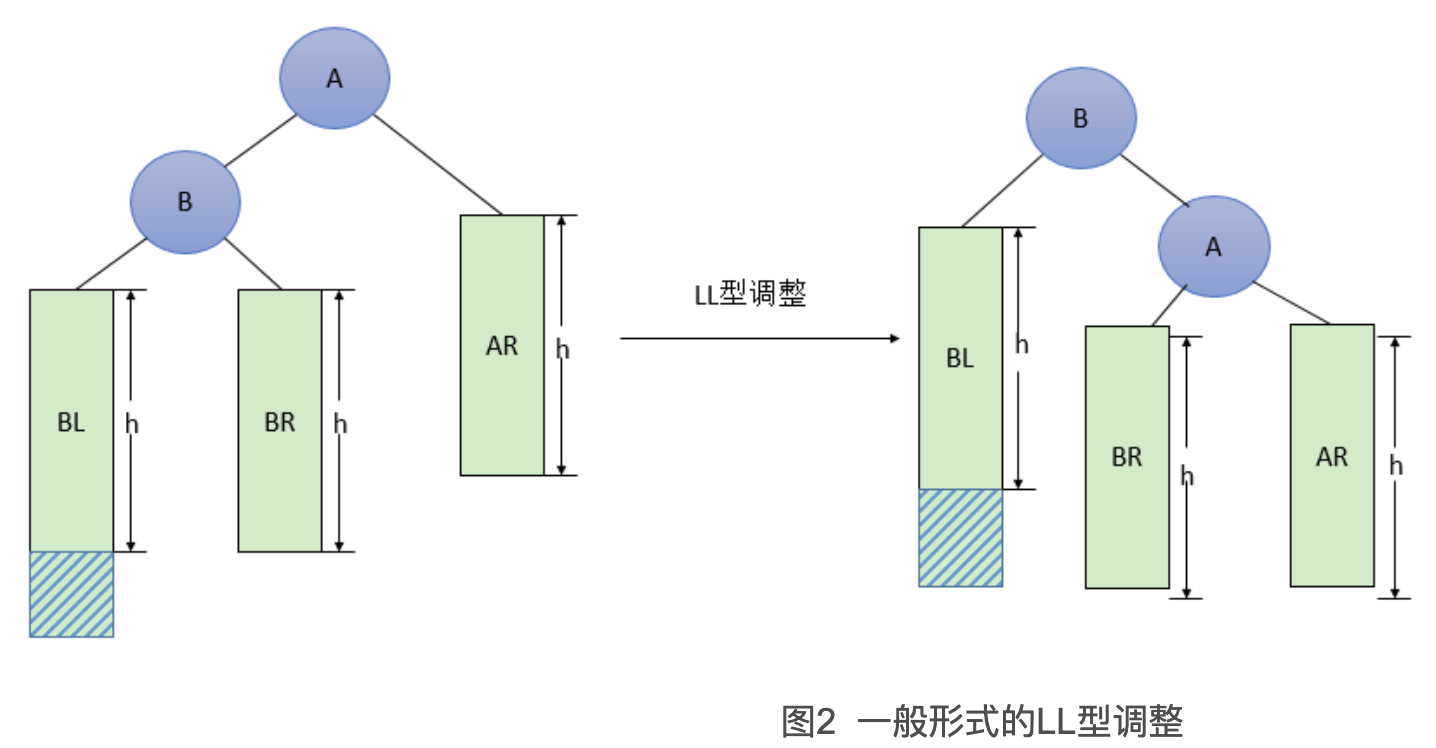
2）**平衡二叉树AVL**：空树或者满足左右子树的深度差的绝对值不超过1，且左右子树都是平衡二叉树的二叉搜索树。

一棵AVL树有如下必要条件：

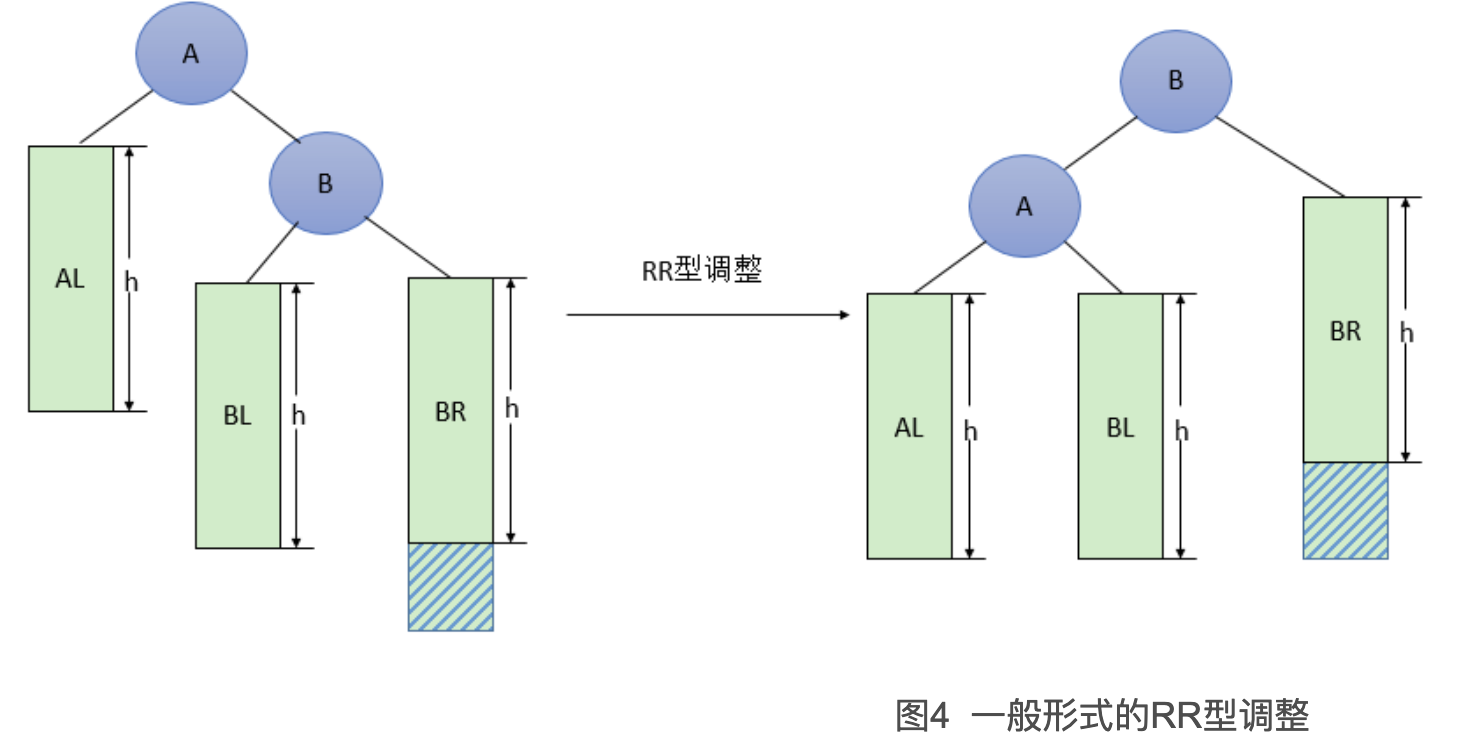
条件一：它必须是二叉查找树。

条件二：每个节点的左子树和右子树的高度差至多为1。

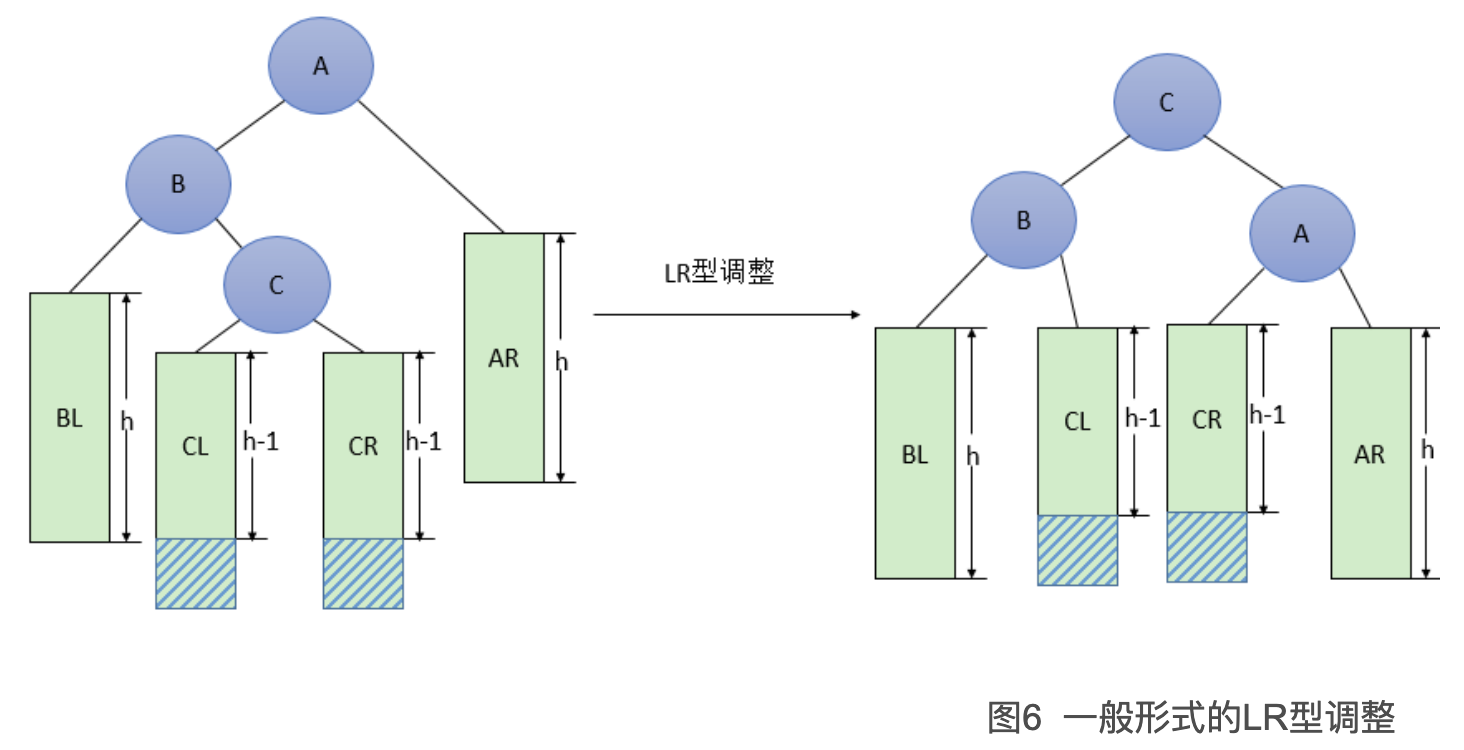
相关概念：a)平衡因子（高度差）b）最小不平衡树（距离插入节点最近的，且平衡因子的绝对值大于1的节点为根的子树）

c）平衡调整：**LL型调整**（由于在左孩子(L)的左子树(L)上插入新结点，使原来平衡二叉树变得不平衡）

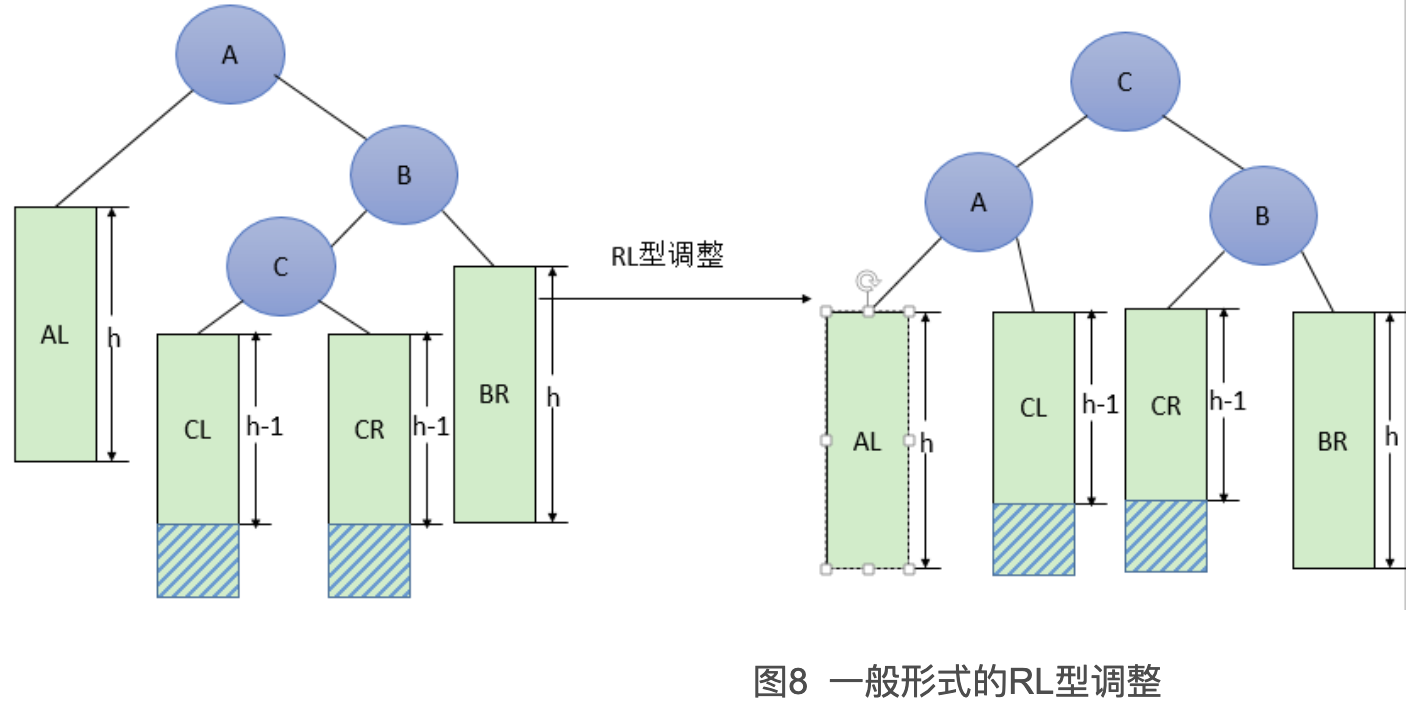
**RR型调整（**右孩子的右子树BR(不一定为空)中插入结点(图中阴影部分所示)而导致不平衡）

****

**LR型**（由于在A的左孩子(L)的右子树(R)上插入新结点，使原来平衡二叉树变得不平衡）



**RL型**（由于在A的右孩子(R)的左子树(L)上插入新结点，使原来平衡二叉树变得不平衡）



维护麻烦

**3) 红黑树：**一种半平衡的二叉搜索树，红黑树中所有叶子节点的深度相差不会超过一倍，它放弃了二叉搜索树的绝对平衡，换来了较为简单的可维护性，使得二叉搜索树插入新数据，以及搜索数据时，都具有不错的搜索性能。

特性：

\* 所有节点要么是红色，要么是黑色

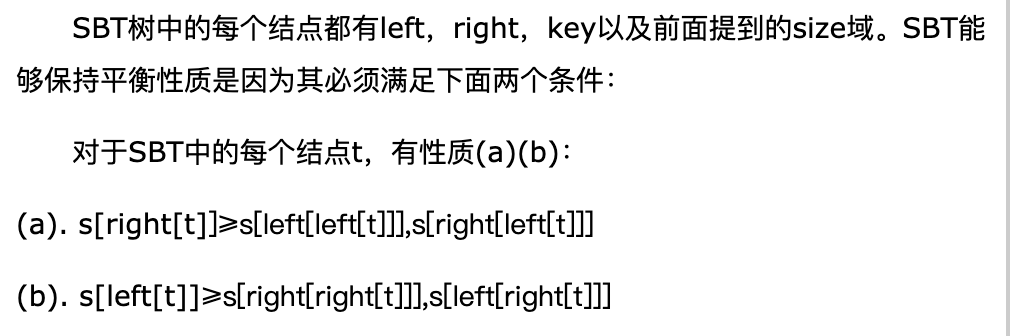
\* 根节点必须是黑色

\* 所有叶子（NULL，or NIL）节点都是黑色的

\* 红色节点的两个子节点必须是黑色（不能有连续的红色节点）

\* 从根节点到任意叶子节点，黑色节点的数目是相等的

**4）SBT**



**5）treap：**二叉查找树和堆的结合，它的核心思想就是给树上的每个节点都取一个随机值，在维护二叉查找树的同时去维护这个随机值成一个大根堆（小根堆）

**6）伸展树splay**

考虑到局部性原理（刚被访问的内容下次可能仍会被访问，查找次数多的内容可能下一次会被访问），为了使整个查找时间更小，被查频率高的那些节点应当经常处于靠近树根的位置。这样，很容易得想到以下这个方案：每次查找节点之后对树进行重构，把被查找的节点搬移到树根，这种自调整形式的二叉查找树就是伸展树。每次对伸展树进行操作后，它均会通过旋转的方法把被访问节点旋转到树根的位置。

1. **哈希冲突解决方式**

1）开放定址法：当发生冲突时，使用某种探测技术在散列表中形成一个探测序列，沿着这个序列逐个单元查找，知道找到给定的关键字或者遇到一个开放的地址，遇到开放定址则表明查找失败

2）拉链法：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组T[0..m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于 1，但一般均取α≤1。

3）再哈希：建立多个哈希表

4）建立公共溢出区：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

1. **TCP长连接与短连接**

当网络通信时采用TCP协议时，在真正的读写操作之前，server与client之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时它们可以释放这个连接，连接的建立是需要三次握手的，而释放则需要4次挥手，所以说每个连接的建立都是需要资源消耗和时间消耗的。

**长连接**是指在一个TCP连接上连续发送多个数据包，在连接保持期间，如果没有数据包发送，需要双方发检测包维持连接。如果给定一个连接2个小时内没有任何动作，服务器向客户端发送一个探测报文，如果客户端的响应正常，继续保持，如果客户端崩溃终止连接。（多用于操作频繁）

**短连接**是指通信双方有数据交互时，就建立一个TCP连接，数据发送完成后，则断开此TCP连接。

1. **TCP拥塞控制**

**慢开始，拥塞避免，快重传，快恢复**

慢开始：刚开始发送方拥塞窗口cwnd的值为1，当发送窗口swnd等于拥塞窗口的值时，即发送一个数据报文段之后，接收方收到该报文段回复一个确认报文段，发送方收到确认后将拥塞窗口的值加倍（2/4/8/16）

拥塞避免：超过慢开始门限之后，拥塞窗口只能+1.如果出现拥塞（超时），更改cwnd和门限，重新开始慢开始算法

快重传：接收方每次对最后收到的报文确认，发送方一旦收到3个连续的重复确认，立即对相应报文段立即重传

快恢复：发送方一旦收到3个重复确认，知道只是丢失个别报文段，此时执行快恢复算法

**流量控制**

**利用滑动窗口** A向B发送数据，建立连接时B告诉A接收窗口的数值，A的发送窗口不能超过接收窗口的数值

1. **同步/异步/阻塞/非阻塞IO**

IO操作：对流（文件）进行数据的收发操作

对于一个套接字上的输入操作，第一步就是通常涉及等待数据从网络中到达，当所等待的分组到达的时候，它就被复制到内核中的某个缓冲区中。第二步就是把数据从内核的缓冲区拷贝到应用进程的缓冲区中

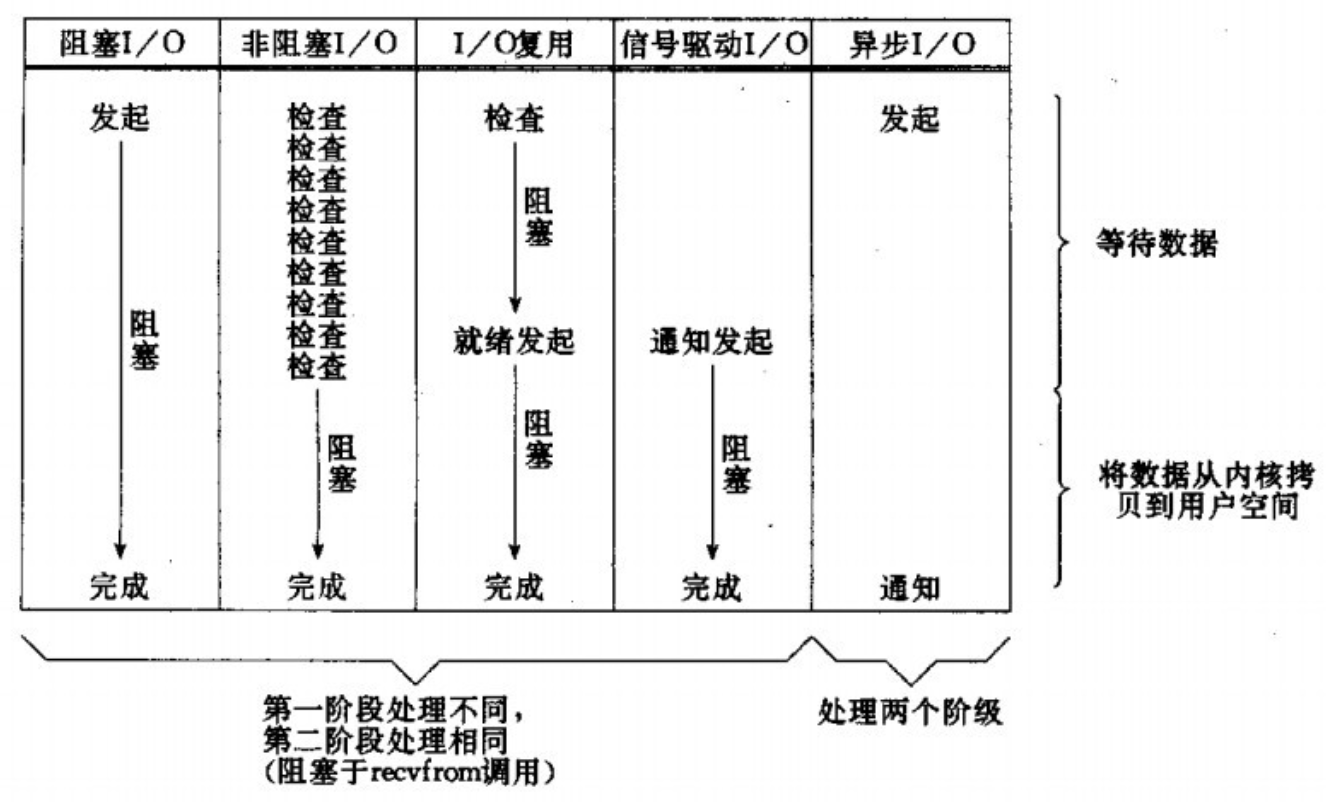
**阻塞IO模型**：等到有数据准备好或者有数据了才继续执行系统调用，最后才从系统调用的函数中返回

**非阻塞IO模型：**通过忙轮询去检测是否有数据准备好，没有数据准备好就一直轮询，直到有数据准备好了可以开始进行数据的复制了为止

**同步**：同步是指用户线程发起IO请求后需要等待或者轮询内核IO操作完成后才能继续执行

**异步**：用户线程发起IO请求后仍继续执行（直接返回），当内核IO操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

**IO多路复用**：允许单线程内处理多个IO请求，但是每个IO请求的过程还是阻塞的（在select函数上阻塞）

****

1. **数据库ACID**

原子性（不可分割的最小单元）

一致性（执行前后保持一致的状态）

隔离性（最终提交前，状态对其他不可见）

持久性（提交后永久保存）

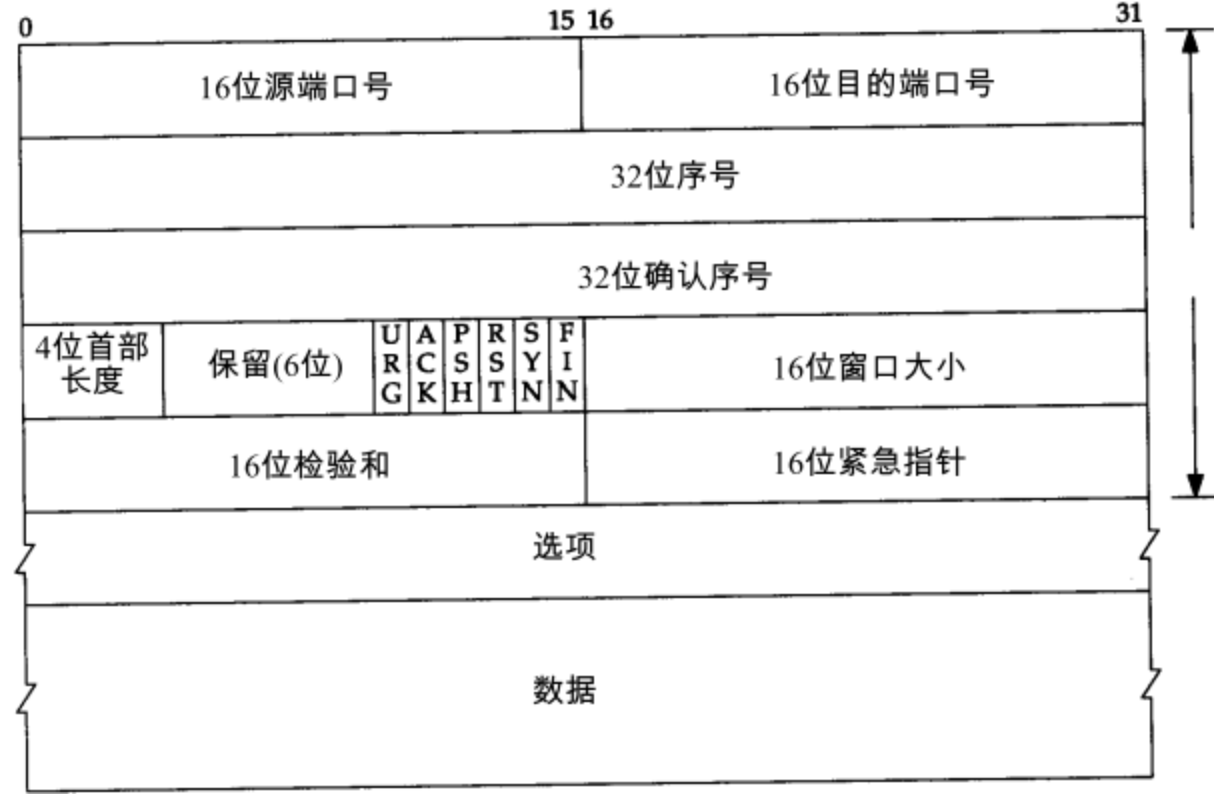
**死锁：**

**互斥，占有和等待，不可强夺，环路等待**

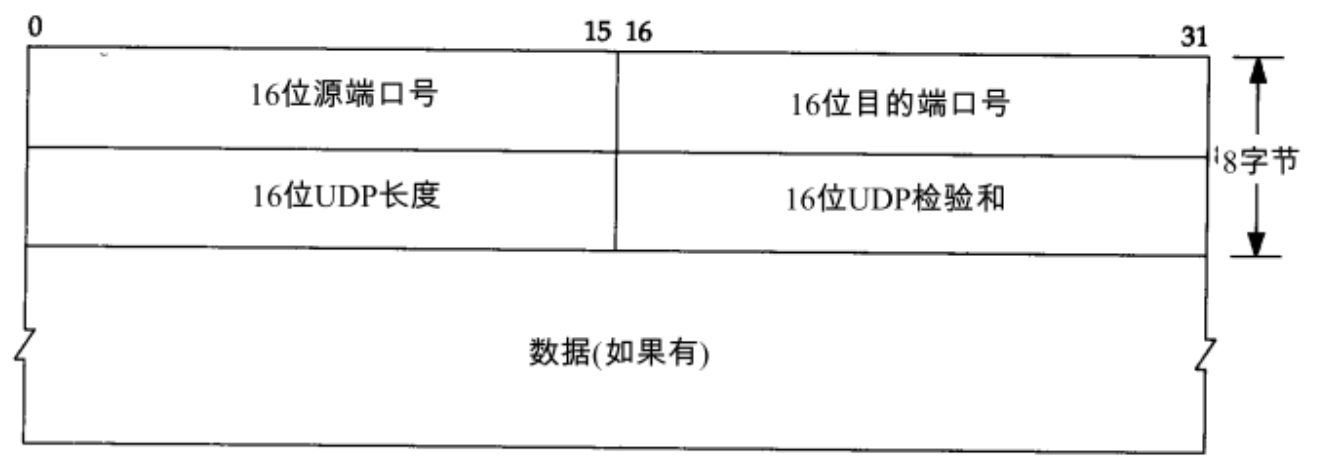
1. **TCP/UDP**

****

TCP头部：源端口，目标端口，序号字段，确认序号字段，首部长度，标志字段，窗口大小，校验和，紧急指针，选项



UDP头部：源端口，目标端口，长度，校验和，数据



**（UDP如何实现可靠传输）**应用层定义一些可靠的协议，比如检测包的顺序，重复包等问题，如果没有收到对方的ACK，重新发包

**游戏**一般使用UDP，因为TCP的特性，一旦丢包就会重发，阻塞后续的数据包，会产生一个比较大的延迟。丢包检查可以使用类似TCP的ask机制，给每个数据包都添加一个sequence ID，然后发送端就依次发送数据包，接收端收到数据包后就可以根据sequence ID来判断是否有丢包了。发送ID的ack时可以附加一个32bit的位序列，表示当前ID之前的32个连续顺位的数据包是否到达。相应在发送端设置一个超时机制。

**TCP的可靠传输机制**

三次握手，四次挥手，拥塞控制

**TCP粘包（UDP不存在）**

发送方发送的若干数据到达接收方时粘成一包，产生原因可能是发送方（TCP默认的Nagle算法，收集多个小分组，在一个确认到达时同时发送），也可能是接收方（收到数据包时没有马上交给应用层处理或者应用层没有立即处理）

使用定长包，使用结构体

**socket TCP编程accept发生在三次握手哪里**

accept过程发生在三次握手之后，三次握手完成后，客户端和服务器就建立了tcp连接并可以进行数据交互了。这时可以调用accept函数获得此连接。Accept正确返回以后TCP Server 可以和Client的连接已建立并可以通信了

1. **浏览器输入url访问过程的优化，设计缓存方案**

过程：1）**http请求过程**，输入地址后，通过域名查找ip，给web服务器发送一个http请求，服务器进行重定向，浏览器跟踪重定向地址，服务器处理请求并返回一个html响应，浏览器解析html并绘制页面，浏览器发送潜入在html中的对象，浏览器发送ajax请求

2）**TCP连接**

3）**缓存处理**：建立了TCP连接之后，服务器可以向浏览器发送资源，可如果是请求过的资源，刷新页面会重新请求资源，造成资源浪费。浏览器对再次请求有缓存策略（强缓存和协商缓存）

**强缓存**：状态码为200，判断缓存的字段有expirs（HTTP日期，根据日期判断缓存是否失效），cache-control和pragma（只有一个属性值no-chche，不用强缓存，优先级更高）

**协商缓存**：请求资源时，把用户本地该资源的 etag 同时带到服务端，服务端和最新资源做对比。如果资源没更改，返回304，浏览器读取本地缓存。如果资源有更改，返回200，返回最新的资源。

1. **SSL与加密方式**

SSL：安全套接层，为网络通信提供安全以及数据完整性，在传输层对数据进行加密，分为记录协议和握手协议。

加密方式：

1）对称加密：使用同一把密钥DES，AES

2）非对称加密RSA，DH，DSA