C++性能优化策略

1 关于继承：

不可否认良好的抽象设计可以让程序更清晰，代码更看起来更好，但是她也是有损失的，在继承体系中子类的创建会调用父类的构造函数，销毁时会调用父类的析构函数，这种消耗会随着继承的深度直线上升，所以不要过度的抽象和继承。

2 对象的复合：

对象的复合和继承很相似，当一个对象包含其他对象构造时也会引起额外的构造。关于这点可能会有很多人不解，认为这是不可避免的，举个例子，你的一个对象中用到数组和字符串，你是选择string和vector还是char\* 和c系的数组呢，如果没有用到c++ stl库提供的相关的高级用法，建议选择后者。

3 构造函数：

尽量用参数列表初始化代替参数，避免值传递初始化。

4 变量延时定义：

从c系转过来的仍保留着c的习惯，在函数第一行先把所有用到的变量都定义好，但是c是没有运行时的消耗的，对于c++是不一样的，对于c++对象的构造和销毁是有消耗的，如果有大量的对象只在某个if条件的一个分支中出现，那就会有50%的情况这些消耗是可以避免的。对于这点在一个类中也是一样的，如果成员中有成员只在某个时刻能用，就用指针代替，在构造对象时初始化成空指针，避免构造时调用他的构造函数。

5 虚函数：

虚函数的底层实现是通过一个虚函数表来实现的，因此有虚函数的类构造是必须先初始化虚函数表，函数调用时也必须先找到虚函数表，然后通过指针偏移找到相应的函数，并且如果虚继承的存在会进一步增长这个过程，它是有运行时消耗的，所以避免滥用虚函数和虚继承，尽可能的用模版设计来代替虚继承把运行时的消耗提前到编译期。

6 返回值优化：

虽然c++编译器会选择性的进行RVO优化但是不是强制的，当函数有多个返回语句并且返回不同名称的对象，函数过于复杂，返回对象没有定义拷贝构造函数时，rvo优化是不会执行的，所以当函数返回一个很大的对象时在不确定rvo优化会执行时，尽量避免值传递。

7 变量的定义：

在定义变量时尽量避免类型的不匹配造成临时变量的产生。

8 内存管理：

c++内存管理的大权由我们自己掌握，对于项目中要频繁申请和释放的对象建议用简单的内存池来管理，可以大大的降低频繁申请和释放内存带来的消耗。

9 善用内联：

内联函数不仅仅是简单的函数调用似的优化，他还有一个最大的优点就是，可以让编译期进行进行边界代码的运行环境优化，内联把代码拷贝到执行环境处避免了函数调用带来的消耗，并且编译期可以进行正常的编译优化，而函数调用是不能实现的。

10 stl ：

记住一点stl不是唯一的选择，有时候也不是最好的选择，合理选择stl善用stl算法。

11 缓存：

对于多次使用的计算结果及时缓存，避免重复计算。

12 延时计算：

对于不关心计算结果的计算过程尽量延时执行或者异步去执行。

13 多线程：

尽可能的使用无锁式多线程开发，锁是一个非常消耗性能的东西，保证数据同步的手段有很多，voalite，原子操作都可已实现，如果迫不得已要尽量减少锁的消耗，比如降低锁的粒度，使用性能更高的锁等等。

14 cpu缓存：

合理的利用cpu cache可以极大的提高代码的运行效率，比如数组中以每列遍历和每行遍历的区别。当然多线程环境下也要考虑它带来的影响。

15 内存对齐：

在进行网络编程时，最好对网络中传送的数据快进行内存补齐，加快网络数据的读区速度。

16 函数参数：

用const引用代替值传递，如果函数参数过多，可以用对象来打包参数，减少参数过多带来的性能消耗。

17 算法:

尽可能的优化你的算法。

18 其他优化方案：

位运算代替乘除法，前缀运算符代替后缀运算等等。