2010 届普通本科毕业论文(设计)　　存档编号:_____

湖北经济学院

毕业论文(设计)

题目: 局域网大文件并发传输设计与实现

专　　业: 计算机科学与技术

院　　系: 信息管理学院

年　　级: 2010 级

学　　号: 1032149

姓　　名: 夏彬

指导老师: 宋莺

职　　称: 讲师

湖北经济学院教务处　制

目 录

摘要

本文设计并实现了一款局域网内的并发传输工具, 可以有效解决同一文件多次传输时的重复发送问题. 并以此工具的实现过程介绍了 linux 下软件开发的常用方式, 利用 C++11 的新特性使代码更加简洁, 使用 python 等脚本语言的优势进行代码生成和测试.

整个软件的开发过程充分利用开源优势, 系统开发与前期测试在 linux 下进行, 主要使用 c++ 进行核心部分的编写, 使用 python 进行测试以及一些辅助开发, 使用 js/css 完成软件页面设计. 在设计时候前端后端有明显分界. 整个软件的代码和论文文档均使用 git 进行版本管理, 使用 cmake 进行项目管理.

关键字: WEB UDP Linux C++11 GIT Python CMake

Abstraction

This paper desgin and write an tool to Transfer file in LAN. …

第 1 章

绪论

1.1 前言

本次设计题目来源于在校期间曾多次遇到教师上课时需要发送一些文件给学生安装以便更好的进行教学安排,但在文件较大时局域网即使处在满负荷之下也无法满足人数众多的学生同时下载。因此萌生使用 UDP 的广播特性来进行文件传输这一想法。考虑到操作系统的多样化以及自己一直是在 linux 环境下成长的,因此跨平台是这款软件的一个必备要求。

本次设计和实现均在 linux 下使用开源工具完成意义在于,在桌面计算机上主流的三种操作系统有 linux、Mac 和 Windows。国内 Windows 在过去占据着主导位置,随着苹果这几年的崛起国内 IT 公司已经开始注重其他系统平台用户。现在国内很多软件都已经拥有 Mac 版本,少部分软件拥有 linux 版本。而且手机、平板这些便携式电脑已经越来越流行,软件平台早已不再局限在 Windows 上。也不会局限在某一特定事物之上。跨平台不仅仅意味着能在多个系统上运行,同时促使软件开发人员在开发软件时尽量不用操作系统特性而是使用更具抽象性的工具如 C++ 的 boost 库。这样能更好的进行抽象设计,而不用过早陷入细节问题。

本次设计主要是基于以太网的特殊性,在物理层每张网卡都会接收到局域网内所有的数据 [7],UDP 本身是不适合进行大文件传输的特别是在互联网上,但合理使用广播或多播这种利用以太网本身的传输机制进行合理的利用带宽,就可以让这种并发传输的情况下非常明显的“突破”网络物理限制。可以节约宝贵的时间,特别是在课堂上。

1.2 本文研究的内容和结构

第 2 章介绍了开发环境以及开发中使用到的一些技术和工具,包括操作系统平台,文本编辑器,编译器,项目管理工具和版本控制等。

第 3 到 5 章讨论了软件的核心设计包括核心数据结构,发送接收过程。

第 6 章介绍了用户界面的选择以及介绍了为更好的使用 WebPage 作为界面而另行设计的一款 HTTP 服务器 AppWebServer。

第 7 章介绍了界面的 js/html 代码部分主要谈论了在使用 WebPage 时遇到的一些问题和选择。

第 8 章简单的介绍了本次测试使用的一些方式。

第 2 章

开发环境

本章介绍了系统中使用到的一些开发工具和系统环境。首先介绍了使用的操作系统同时稍微比较了一下 linux 和 windows 的各自优缺点。然后列举了使用的开发工具并没有使用 IDE 进行开发,IDE 是一个很大的进步但亦有其缺点,并且命令行下的构建工具也是在发展。这里介绍的工具和 windows 下的开发方式有较大区别本文只简单介绍使用的工具不过多的阐明原因,对于长期使用 IDE 的朋友可以选择看一下云风的一篇连载文章 “IDE 不是程序员的唯一选择”,这里并非争论谁更高效,只是希望每个人都能有机会进行选择。

2.1 系统环境

开发环境是 archlinux 系统 (其他 linux 系统同样适用),Arch Linux 的官方介绍很简短:

> “A lightweight and flexible Linux® distribution that tries to Keep It Simple.“

将近 5 年的日常使用感受是它很符合官方介绍所说的 KISS 原则,保持 linux 的简单和灵活。它不像其他发行版有固定的版本号,archlinux 使用 pacman 和 aur 进行滚动升级,类似 gentoo 的管理方式,但常用软件官方有提供二进制包。可以让你在第一时间使用到最新的软件版本。

windows 和 linux 的一个很大区别,windows 很多工具都比 linux 下要更加方便 (但一般都是收费软件且价格不菲)。windows 下由于主要是靠商业程序来丰富应用因此它上面的程序一般会比较好用但于其他软件的交互上基本已经让用户错误的认为程序间不需要什么交流。linux 下则不同大部分软件虽然完成的功能很单一接口也

不是很友好,但十分容易和其他工具结合起来用,用户可以任意选择各种工具组合起来,可以在遇到好用的工具后将其集成到自己的工作环境中。linux 下大部分软件的接口都会有其常用的约定,比如 -s 以便代表大小 (可能是指示显示文件大小或者是按照文件大小排序);-h 一般是指 human 表示使用较友好的输出格式或者是 help 的意思。有时候对于第一次使用的工具会在没有看帮助信息的情况下自然的使用到正确的指令 (当然仅仅是非常通用的)。

在 linux 你可以把绝大部分重复的工作自动化,比如在编写这款软件的时候需要在虚拟机里面测试 windows 下的运行情况,需要来回的拷贝编译好的文件到虚拟机去执行。在厌烦最初的几次拷贝后便编写了几行脚本来完成这个工作,现在只需要进入 WIN32 目录下 make 一下就可以自动编译最新的版本且自动将文件拷贝到 windows 中。在切换进 win 系统后就可以直接运行了 (当然也比较容易做到 make 之后让 win 下最新的版本自动运行)。

linux 也有其不方便的地方比如桌面软件很多情况下需要考虑 win 下用户的使用造成调试繁琐;它的图形接口在稳定的前提下没有 Windows 友好;国内大部分商业公司的软件均不支持 linux;完成同一件事情会给你过多的选择 (虽然这一点也可以算作优点),造成刚刚进入某个领域时对于选择什么工具或技术十分困难。

这里选择 linux 仅仅是因为长期使用的原因,我个人是很少学习 linux 系统本身的 API。因此不会使用 linux 本身的特性,尽量保证程序能做到平台无关。

2.2 开发工具

2.2.1 文本编辑器

在 windows 下学习编程的同学可能很少有文本编辑器的概念,我在接触 linux 之前也只了解记事本这个简陋的软件;Word 这种处理文档的工具;VB6.0、Dreamware 等集成开发环境。

现在编程语言如此之多,对于一个稍有好奇心的人来说都仅仅使用一种语言是残酷的。因此如果说现今什么编辑工具最可能支持所有的语言那肯定是非 vim 和 emacs 莫属了。这里选择使用 vim 作为代码编辑工具原因很简单它是一个优秀的文本编辑器而,并且也只熟悉这一个文本编辑器。

在软件编写和论文撰写阶段至需要编辑 python 代码、C++ 代码 (且需要支持刚刚

发布的 11 标准的部分新语法和工具)、html 代码、Javascript 代码、CSS 代码、latex 代码以及 CMake 的配置文件。这仅仅只是在完成这个简单的毕业论文需要应付的语言。而在日常学习生活中需要遇到的文件类型更是各种各样,仅仅我个人平常需要编辑的文件种类都可以继续列出许多。

如果每种文件都使用特意为其编写的工具来编辑那么在计算机这个日新月异的发展速度下你在学习某种技术的时候就需要花费一些不必要的额外时间。而使用 vim 这类工具你需要的仅仅是在遇到一种新语言或文件的时候在网络上花上十多分钟就能获得不错的效果。除了少数通用约定 (如 ctrl+v,ctrl+z,alt+<F4> 之类) 你在一个工具上积累的知识很可能在下一个工具上就不再适用了。而使用文本编辑器你可以解决任何纯文本格式的编写问题,你所积累的会越来越多你的效率也会随之提高。因为其设计理念它不会随着时代的变迁而变的不再适用 (至少 vim 的前身 vi 的年龄比我要大)。

vim 和 emacs 在网上已经有很多中文资料我在此就不在这里介绍了。

如果没有 vim 这样的文本编辑器则在本次论文过程中就很难处理众多的文件格式。

2.2.2 编译器

C++ 的编译器种类并不多成熟的只有 GCC,VC,Intel C++ 以及新秀 clang(Apple 主导研发的基于 LLVM 的编译器)。其他如 TrubC++、Borland C++ 由于失去支持我觉得即使作为教学也不该使用。Intel C++ 口碑不错但是商业程序,VC2010 有 express 版但只能在 win 下使用。clang++ 对 c++11 的支持不够多,因此这里选择 gcc4.6.2 作为主要的工具。win 下的编译工具使用 mingw。win 下面使用在发布的时候选用 vc++ 进行编译也许是更好的选择。

在交叉编译上这里介绍一个叫做 mingw cross 的项目。这个项目主要是帮助用户在 linux 下建立一整套完整的交叉编译工具 (仅仅针对 win32) 和常用的库。通过脚本进行构造最简单的情况下可以仅仅下载完脚本文件一个 make 命令后自动下载源代码自动编译大部分软件库 (比如 qt,gtk 等),如果手动编译的话会有很多问题而这个项目通过测试特定版本对有问题的地方给出 workaround 的 path 从而大大简化了交叉编译。

2.2.3 构建工具

这里使用 CMake 进行构建管理。CMake 通过读取配置文件 CMakeLists。txt 来生成 makefile 文件。可以方便的进行工程管理。较老的资料都是介绍 GNU 的 Autotools 系列工具,只是这套工具还是稍嫌麻烦,当然其历史悠久稳定可靠。但如果不是为了开发完全符合 gnu 规则的软件则使用 CMake 是十分好的选择。另外 CMake 可以生成其他 IDE 需要的工程文件比如 VS。

如果使用 Java 语言则 Ant 是最好的选择了。还有两款使用 python 写的 C/C++ 构建工具 SCons 和 Waf 也有一定名气。

本次使用 CMake 来管理项目使编译过程变得十分轻松,不论是交叉编译 windows 下的可执行文件还是 linux 下的可执行文件都只需使用同一份 CMakelists。txt 进行管理一个 make 命令就行了。

2.2.4 版本管理

这里仅仅讨论开源的版本管理软件。Git 是较新的版本管理软件 05 年才刚开发出来,而我也只是这几年才了解。Git 作者对其名称来源是这样解释的:

> "I'm an egotistical bastard, and I name all my projects after. First Linux, now git."

从 First Linux 就知道这句话是 Linus Torvalds 说的了,他也是 git 的最初作者。

因为早期使用的商业软件 BitKeeper 合作到期 Linux 社区急需一款优秀的版本管理软件来管理日益庞大的 Kernel,所以创造了 Git。

这之前最流行的开源 CVS 是软件要属 SVN 了,著名的 sourceforge 就是使用 svn 进行代码托管的,Google Code 也是,国内的 Sina App Engine 也是。我最初使用的 CVS 也是 svn,但 svn 需要开启一个服务进程稍显麻烦。CVS 虽然是 Concurrent Versions System 的缩写但它同时也是一款历史悠久的版本管理软件的名称。

使用版本控制几乎已经成为软件开发中必须的一个环节了,特别是 git 的出现是部署版本管理的成本几乎为零,你所需要做的仅仅是在命令行下敲入 git init 就行了。在编写代码时候就多次利用 git 的分支功能进行一些想法上的实践,如果没有版本控制就必须出现一个想法的时候把源代码复制保存一份以免破坏原有代码。

2.2.5 文档工具

这里使用的 latex 是指编写本论文的工具,而非软件文档。如果所写程序属于供他人使用的代码库则可以用 doxygen 自动生成文档。本论文很少涉及到数学公式而且没有现成的模板文件因此使用 latex 并非是一个很好的选择,但考虑到 linux 下使用 libreoffice(基本可以认为是 OpenOffice) 会在一些细节地方与 MS Word 不兼容且自身对 word 类工具使用经验较少。因此选择 latex 这类用户不需要关注版式但却能很自动排版的工具。其中中文宏包使用 xeCJK。利用 latex 使本次论文的编写不需要关注论文版式上的问题[1]。

[1]当然在没有模板的情况下还是需要花一定时间去调整版式,因此特意将完成的 latex 模板放在 github.com 上以便以后的同学使用 latex 编写本校论文不需要关注论文格式。

第 3 章

核心数据结构

由于 UDP 缺乏可靠性 [1] 可能出现丢包、出错、复制等现象，并且缺乏拥塞控制机制 (congestion control)。对于丢包、出错等现象对于需要绝对安全的文件传输来说是绝对不能允许的，对于大文件的传输如果没有拥塞控制则可能出现大量丢包 (通过之前做本地回环实验丢包率可达到 30%，实际局域网不会这么高)，或者由于传输间隔过长导致效率急剧下降。

因此必须设计特定结构来克服这些困难。

3.1 散列函数

散列函数 (Hash Function) Hash Function 可以从任何一种数据中创建较小的数字 “指纹”，它可以把很大的数据量压缩成较小的固定位数的数字 [10].

这里 Hash 代表系统的一个数据结构 struct Hash; 通过使用 Hash 结构来对某一事物进行唯一标示。准确的唯一标示应该使用 UUID，但 UUID 只是产生一个唯一标示符但不能和对应的事物产生关联。

在表示文件时使用 Hash 进行唯一标示以免文件名相同但内容不同的文件出现。在传输数据的时候使用 Hash 对传输的数据块进行完整性验证以保证最终收到的文件不会有任何字节偏差[1]。

最出名的 Hash 算法要属 MD5，MD5 常用来对用户密码进行加密 (稍微安全一些的做法是加上 salt)。另外还有 SHA-0，SHA-1，SHA-256，SHA-512 更安全的算法也较为出名。

考虑到计算量以及允许出现碰撞情况的出现，这里没有选用上面这些较好的 Hash

[1]传输二进制文件必须保证没有任何字节出错。

算法,而是使用计算量较低的 MD4,没有使用 MD2 的原因是 MD2 使用 16 位整型进行操作在 32 位或者 64 位 CPU 上不一定会带来速度的提升。

源码中仅仅使用 typedef 将一个 string 作为 Hash 类型 typedef std::string Hash; 具体计算 hash 是通过 Hash hash_data(const char* data, size_t size); 函数调用crypto++库的 md4 函数进行计算。

3.2 文件的抽象表示

在有操作系统和文件系统的支持下我们可以方便的使用文件路径进行文件定位以及操作。而进行网络传输时则需要把文件的一些必备信息告诉对方这里涉及到文件大小,文件有多少块,一次发送整个文件是不理智的 UDP 协议也因为 length 字段仅仅 16 位 [1] 所以一次最多只能传送大约 64Kb 的数据,一块数据有多大,文件名和真实路径,文件类型等问题。

这里先将系统使用的数据结构给出

```
struct FInfo {
        enum Type : uint8_t {
                NormalFile, Directory, RootFile, RootDir
        } file_type;
        enum Status {
                Local, Downloading,Remote
        } status;
        Hash parent_hash;
        Hash hash;
        uint32_t chunknum;
        uint16_t lastchunksize;
        std::string path;
}
```

enum Type 使用了 c++11 的新特性 [6] 用来指定 enum 底层的数据类型[2],因为C/C++的基本类型并非固定大小因此在进行网络传输时必须使用定长类型或

[2]默认为 int。

者对变长类型[3]进行特殊处理。uintXX_t 在 c++11 中是作为标准类型出现的 (定义在 cstdint 中)。enum Status 由于不需要在网络中进行传输仅仅是在本地进行状态变迁使用的,所以无须指定底层类型。

FInfo 可以通过 info_to_net 和 info_from_net 这一组函数在网络上进行传输。chunknum 表示文件有多少块数据; 普通文件块的长度是一个在 config.hpp 中定义的固定值[4] lastchunksize 表示最后一块数据的长度[5]。通过这 2 个变量和一个常量就可以计算出某个文件的大小以及文件块数。

path 在不同情况下意义稍有不同,在文件正在传输时 path 指向了操作系统中对应文件的真实路径[6]。刚通过网络接收后但还没开始进行下载时 path 仅仅保存了文件的名称以便用户大概知道文件的内容。由于 path 字段是变长大小因此在进行传输时需要进行特定处理,这里使用的方式是用一个 uint8_t 代表 path 长度,在进行传输和接收时自动处理。因此 path 的长度固定在 1 256 之间,超过这个长度系统会自动截断但会优先保留后缀名以便用户有机会通过后缀名初步判断此文件是否需要。

hash 字段是通过对整个文件进行 hashing 得到的 16 位定长数据,可以很好的用来区别不同文件。

status 字段是用来表示当前文件状态的,刚从网络接收到的 FInfo 都是 Remote 状态,开始下载后转换为 Downloading 状态,下载完成后成为 Local 状态。如果 FInfo 是本机制造的则为 Local 状态。

以上几个字段就可以很好的处理普通文件传输了,但如果需要传输目录则需要进一般处理,这里增加了 file_type 和 parent_hash 两个字段来完成。共有 4 种文件类型普通文件 (NormalFile),目录 (Directory),根文件 (RootFile),根目录 (RootDir),其中 RootFile 和 RootDir 不需要 parent_hash 这个字段 (因此在进行实际传输时会根据文件类型来判断是否发送或接收 parent_hash)。为了较容易理解这里举出一个例子: 假如使用 data 作为路径来建立 FInfo 则会产生以下 7 个 FInfo 出来。

[3]path 字段就是变长的。
[4]当前通过考虑选择 60000。
[5]文件长度不会总能被 60000 整除。
[6]上传者和下载者一般是不同路径的。

```
|-- data   //RootFile
|   |-- cover.tex    //NormalFile  -> data.hash
|   |-- hbue.cls //NormalFile  -> data.hash
|   |-- images //Directory   -> data.hash
|   |   |-- badge2.png  //NormalFile  -> images.hash
|   |   `-- badge.png  //NormalFile  -> images.hash
|   `-- report_imp.tex //NormalFile  -> data.hash
```

其中->指向的是parent_hash

这里仅仅是在进行普通文件传输的原基础之上改造了一个传输目录的方式,但并不建议使用目录传输方式,特别是这里设计的方式有大量冗余信息。因为目录传输的缺点在于文件数量非常多的时候速度会急剧下降建议使用 tar,rar,7z 等能进行打包[7]后传输。这种目录传输方式有一个好处是用户可以有选择的仅仅下载目录中部分文件,并且 FInfo 不需要一次性传输,可以在打开 RootDir 的时候仅仅获取下一级的目录,这样就类似于 AJAX 技术,在不影响用户体验的时候减少不必要的传输。

3.3 文件完整性和分块

如果传输的是视频、音频等文件丢几个字节没有很大关系,这也是使用 UDP 的主要原因,但很可惜本系统无法利用 UDP 的这些优点,必须保证文件的完整。UDP 会出现丢包、重复、乱序等问题。但都可以通过分块传输来解决。数据结构如下:

```
1 struct Chunk {
2         Hash file_hash;
3         Hash chunk_hash;
4         uint32_t index;
5         uint16_t size;
6         const char* data;
7 }
```

[7]在 unix 下很常见的一种方式并非是压缩,处理速度很快。

Chunk 表示一个文件块; file_hash 表示此文件块所属文件; chunk_hash 是此文件块所携带数据的 hash 值; index 表示此文件块在整个文件中的序号; size 表示所携带数据长度; data 在发送时携带 size 长的真实数据,在发送前仅仅指向数据地址。

这里同样使用的都是定长类型,以便在不同操作系统下可以顺利完成通讯。其中 chunk_hash 在发送前进行 hashing,接收者在接收到一个文件块后如果通过计算发送所携带数据和 chunk_hash 不符则直接丢弃。

顺便说明这里能够表达的最大文件大小并非 $2^{16} * 2^{32}$ 一是因为实际大小和文件系统有关,二是在下面要介绍的数据结构 Bill 中也会有限制。

3.4 文件块的请求

因为是分块发送的,所以请求者无法直接发送一条类似 GIVE_ME_FILE_XXX 这样的指令到网络然后就等着接收文件[8]。因此需要使用特定的结构来进行特定文件块的请求。这里使用 Bill 结构,示意代码如下:

```
1 struct Bill {
2   Hash hash;
3   uint16_t region;
4   BlockType bits;
5 };
```

hash 表示请求的文件 hash; region 表示当前区域; bits 表示此区域内的请求信息。

BlockType 在 config.hpp 中定义为 typedef uint32_t BlockType 是一个 32 位无符号类型。bits 中的某一位若为 1 表示需要此块文件若为 0 表示不需要。这里因为需要进行网络传输因此使用了 uint32_t 结构,但实际操作时是转换为 std::bitset<BLOCK_LEN> 类型来方便位操作。由于这个结构只提供通讯的辅助信息但又是需要大量发送的,所以需要尽量高效的利用空间。在 bits 和 region 的类型选取上进行了一定考虑。BlockType 不能太小否则会蜕化为一次请求一块的效果了。但也不能太大否则就成了一次这里选取 region 为 16 位可以表达 65536 个区域,每一个区域可以携带 32 块信息,每一个最多携带 60000B 数据 (BLOCK_SIZE) 因此可以表达

[8]UDP 的丢包、重复、乱序等造成。

出 $32 * 2^{16} * 60000 \approx 1171875GB$ 大小的文件。整个 Bill 占用空间 sizeof(Bill) = 16 + 16 + 32 刚好等于 64，在 32 位操作系统和 64 位操作系统下都已经是对齐的结构。

3.5 丢包率

本系统中由于 Bill 的数据量较小在局域网中不会造成太大影响。因此这里只考虑大量发送 Chunk 时的情况。TCP 使用的拥塞控制算法 [2] 有 “慢启动”(slow start)，“拥塞避免”(congestion avoidance)，“快速重传”(fast retransmit) 和 “快速恢复”(fast recovery) 具体见 TCP/IP 详解 [7]。TCP 的这些算法很多是面向环境恶劣的 Internet 网络，而这篇论文所讨论的系统只需要处理合理控制好发送间隔就能很好的处理问题。

处理方法仅仅是根据一个 interval(int 类型) 的值来在发送 chunk 的时候进行一定时间的延迟，这个值的计算是通过整个网络的丢包率进行动态计算的大小在 0~32 之间，

$$\text{丢包率} : \omega = \frac{interval}{320} * 100\%$$

$$\text{间隔时间} : i = \omega * 40ms$$

这里选用 40ms 作为最大值是通过粗略估计算的出:

设 M 为网卡带最大带宽，单位 byte/;，

则在满带宽情况下计算出每 1byte 需要使用的时间 $s = \frac{1000}{M}$，单位 ms;

然后计算出发送一块文件块的时间 $t = s * 60000$; 采用典型的网卡设备 100Mbps 和 1Gbps 分别计算得出 4。6ms 和 0。46ms 的结果，再用同样的方式计算硬盘读取的速度，假设硬盘读取速度 30Mb/s，算出读取一块数据需要 2ms。即使再加上 CPU 处理时间一块数据的处理时间不会超过 10ms。实际代码中接收端是通过多线程写入数据只要收到数据后可以忽略写入时间。因此 40ms 在局域网已经是非常保守的值了，初始状态 interval 的值是 0。具体如何计算 interval 需要设计到其他结构因此在下一章的 “速度统计与丢包率” 中进行介绍。

第 4 章

文件发送与接收

本章介绍先后尝试的三种传输模型:“一问一答”、‘并发处理”、“多此迭代”。最终通过分析综合考虑选择使用 “多次迭代”。

这一章由于涉及到的概念并非十分具体因此为了更好的表述做以下约定:

- S 为发送者。
- R_x 为接收者,如 R_1R_2 等。
- 涉及的文件拥有 N 块数据[1]。
- 当前发送的文件块[2]为 $C_x(0 < x < N)$。
- 文件块请求以 B_x[3]为一组,一组包含 0 ~ 32 块请求 $0 < x < N/32$。
- 最大文件区域的值 $n = N/32$

所有文件传输模型均只考虑一对多的方式即一个发送者多个接收者。

4.1 一问一答

采用请求发送模式,R 发送需要的文件块到网络中,S 收到后发送文件块到网络。但由于是一对多的发送方式,一份文件可能有多个接收者,导致 S 重复发送文件块。

例如,一个节点 R_1 开始请求数据后,文件传输途中另外一节点 R_2 发送请求通讯过程如下:

[1]FInfo.chunknum 字段的值。
[2]即之前介绍的 Chunk 结构。
[3]与之前介绍的 Bill 有关。

1. R_1 发送 $B_1B_2\ldots B_n$ 到网络中;

2. S 收到 $B_1 \sim B_n$ 后开始发送 $CB_1CB_2\ldots CB_i(i<n)$;

3. 在 S 未发送完毕时,R_2 开始请求下载,发送 $B_1B_2\ldots B_n$ 到网络中;

4. S 在发送完 CB_n 后继续发送 $CB_1\ldots B_i$,此时 R_1R_2 都已经拥有完整的文件拷贝;

5. S 继续发送 $CB_{i+1}\ldots CB_n$ 到网络中,此时所发数据会无故占用网络带宽;

4.2 多次迭代

因此不能使用这种简单的一问一答的方式发送文件块,必须设计一种 “智能” 的发送方式。

把种方式称做 “多次迭代” 是因为 S 只处理比 PCR 值大的文件块请求,小于这个值的请求则直接忽略直到没有文件块可传则结束本轮传输。

对于发送者 S 制定如下规则:

- 维护一个当前文件发送状态 STATE 取值范围: BEGIN、SENDING、END STATE
  默认状态为 END;
  在 END 状态下 R_x 收到 SB 命令 S 转换为 BEGIN 状态;
  在 BEGIN 状态下收到 B_x 转换为 SENDING 状态;
  在 SENDING 状态下发送 B_n 之后转换为 END 状态;

- 维护一个指向当前发送文件区域的指针 PCR(Point to Current Region),取值范围:$0 \le PCR \le N/32$;
  PCR 的值仅在 SENDING 状态下有效;
  PCR 的值根据收到的最大 B_x 设置,如先后收到 $B_1B_2B_3B_8B_4$ 则 PCR 的状态先后为 1,2,3,8,8;

- 在收到 B_x 后若 x<=PCR 则忽略此 B_x 请求;

- 在 END 状态下忽略一切 B_x 请求;

- 从 SENDING 转换为 END 时候发送 SE;

对于接收者 R_x 制定如下规则:

- 请求开始时发送 SB 指令后发送 $B_i \ldots B_j$ 到网络,其中 $0 \leq i \leq j \leq N/32$;
- B_x 请求顺序必须按照升序排列;
- 在收到所缺区域中值最大的区域时结束请求。例如所缺文件区域为 $B1B2B3B4$,当前已经接收到 $CB1CB2$。如果此时接收到 CB_4 则直接结束,虽然 CB_3 暂时未收到;
- 在收到 SE 的时候结束当前请求;
- 在结束请求后若文件未下载完成则重新开始请求;
- 发送的 B_x 序列必须包含 B_n 若不需要此区域的文件块可发送空的 B_n;

按照设定的规则之后,再来看之前遇到的问题:

1. R_1 发送 $SB, B_1B_2 \ldots B_n$ 到网络中;
2. S 在收到 SB 后从 END 状态转换为 BEGIN 状态,收到 B_1 后从 BEGIN 转换为 SENDING 状态,并设置 PCR=1,在收到 B_2 后设置 PCR=2,...设置 PCR=x;
3. 在 S 未发送完毕时,R_2 开始请求下载,发送 SB,$B_1B_2 \ldots B_n$ 到网络中;
4. 由于 S 在 SENDING 状态忽略 SB 命令,收到 B_1,小于 PCR=x 忽略,B_2,小于 PCR=x 忽略;
5. S 在发送完 CB_n 后 PCR=n,因此忽略队列之后的请求,此时 R_1 拥有完整拷贝,R_2 拥有 CB_x 之后的拷贝;
6. R_1 收到 CB_n 后完成下载退出通讯;R_2 则结束当前请求发送 SE,S 收到后状态转化为 END;
7. 由于 R_2 结束请求后检测还有未完成的文件块,因此重新开始建立请求,发送 SB,$B_1B_2 \ldots B_{x-1}$;
8. 此时网络退化为点对点传送[4];

[4]如果之前有 n 个 R_x 则根据退出数量 y 蜕化为 1 对 n-y 传输。

不过发送 B_x 的速度很快,例如一个 1G 的文件根据系统的设定 Chunk-Size=60000b,一个 Bill 最多可携带 32 块 Chunk,一个 Bill 的大小是 16+16+32=64b,因此所需数据量是,$(1G/60000b/32) * 64 = 35791b$ 只有 30 多 Kb 在局域网内相对于人的操作是瞬间完成的。因此更多的时候是这种情况: S 收到的数据是这样

$$SBB_1B_2 \cdots B_n$$

$$SBB_1B_2 \cdots B_n$$

$$SBB_1B_2 \cdots B_n$$

则 S 对应的处理序列是: 发送 $CB_1 \cdots CB_n$ 后忽略之后队列中所有的 B_x 和 SB 此时大部分 R_x 都已经在收到 CB_n 后完成下载。

但由于丢包的原因可能导致部分 R_x 还有少部分数据块未接收到。这些 R_x 在收到 SE 后结束当前请求,并根据下载规则重新请求下载。之后 S 收到的数据是这样:

$$SBB_{x1}B_{x2}B_{x8}B_n(R_1)$$

$$SBB_{x2}B_{x7}B_{x8}B_{x10}B_n(R_2)$$

$$SBB_{x3}B_{x4}B_{x6}B_n(R_3)$$

$$x1 \leq x2 \leq x3 \cdots xnx1x2$$

S 会根据之前的规则处理 $SBB_{x1}B_{x2}B_{x8}B_{x10}$ 其中后面的 SB 和 $B_{x7}B_{x3}B_{x4}B_{x6}$ 等都被忽略。R_1 在收到 B_{x8} 后完成下载; R_2 在收到 SE 后发送 SB,$B_{x7}B_{x10}B_n$ 到网络; R_3 在收到 SE 后发送 SB,$B_{x3}B_{x4}B_{x6}B_n$ 到网络;
此时 S 收到以下序列:

$$SBB_{x7}B_{x10}B_nSBB_{x3}B_{x4}B_{x6}B_n$$

处理完毕后 R_2 收到所有的文件块结束下载,R_3 依然没收到任何数据,但收到 SE 后重新发送 SB,$B_{x3}B_{x4}B_{x6}B_n$ 到网络中,此后完成所有发送过程。

以上过程发生概率虽然不算罕见,但发生时所缺少的 B_x 量一般都会很小因此虽然处理过程复杂但会很快完成。及时在最坏情况下也会至少处理一个 B_x 请求。而局域网内一般只会几十人下载收敛速度会非常快。并且其中 R_3 的数据是特意构造导致前

几次的迭代过程中没有收到任何有效的文件块,但如果分析实际情况,R_3 所缺少的文件块很可能与其他 R_x 大部分相同。原因在于所缺少的文件块部分是因为丢包造成的。

如果以上过程中发送的 B_n 因为网络原因丢包了,如果是 R_1 的 B_n 丢包了则会加快处理速度,因为 R_2 的 B_{x10} 可以不用等待下一次迭代就发送完毕。如果是 R_2 或者 R_3 的 R_n 丢包了则会导致 S 和 R 都停止工作。

因此需要保证 S 在无事可做时尽快进入 END 以便通知 R_x 此轮请求结束,同时也需要保证 R_x 有接收到 S 发送的 SE 指令。否则可能导致 S 和 R 都在等待对方的信息。为了防止这种情况的发生,S 和 R 都有一个定时器,S 在 SENDING 状态下一定时间内若没有发送任何 CB_x 则自动进入 END 状态。R 在一定时间内没有收到文件块也会自动结束当前请求。

对于文件的发送接收有一个重要原则,宁可让一个 R_x 重新发送全部的 B_x 也不能发送一块无效的文件块,因为一个文件块就有 60000B,而且一般一次发送 32 块,即使按照平均长度一次无效的文件块发送也会导致 960K 的浪费。而前面有计算过 1G 的文件也只有 30K 的 B_x 数据。而造成重复 B_x 的原因是发送 B_x 的数据量小导致发送速度非常快,往往一个 CB_x 都还没发送完毕所有 R_x 的 B_x 就已经进入到了 S 的处理队列中,因此 S 的队列中有非常多的重复 B_x,必须使用一个指针来指名当前位置,到达最后位置的时候就将队列中所有数据全部抛弃 (绝大部分都是重复无效的数据) 让 R_x 重新发送请求。

4.3 并发处理

之前还尝试过另外一种模式,使用一个 map 记录当前系统所缺少的文件块 (根据 B_x 来维护这个 map),则如果接收到重复的 B_x 后由于已经有记录了就直接抛弃处理。这种模式可以很好的解决部分问题,但在实现中必须使用多线程,发送线程和处理 B_x 线程且需要共享这个 map 结构,较为麻烦。且同样需要加入超时机制防止 R、S 停止工作,重新设计一种方案,重新请求由于丢包导致 R_x 没有接收到的文件块。可以使用超时重传但效率过低。而上面讨论的方案由于有多个 B_n 的冗余保证了快速重传 (只要不是最后一个 B_n 丢包了则不会影响效率相反由于 B_n 的丢包导致之后的 B_x 可以继续发送 (需要大小 PCR 的 B_x);

两种方案都可以尽量减少错误的重发文件块,前者通过消耗更多的流量来达到重

传机制,后者虽然不消耗多余的流量但需要消耗宝贵的时间。但在局域网中宝贵的资源是带宽而不是流量,如果使用超时机制来实现重传则其实在浪费时间的同时浪费了大量的带宽,因此在最初选择后者之后改为选择前者来实现本工具。

第 5 章

系统其他部分

本章未完成

5.1 速度统计与拥塞控制

速度统计分为以下三种

1. 局域网内全局有效下载速度;
2. 单一节点有效下载速度;
3. 单一节点特定文件有效下载速度;

其中后两者可以只需要根据单节点自身统计信息就可计算出来

PB 指向当前还未验证的文件块头, PM 指向已经

count([PM,PB]) = THRESHOLD/2 一次最多发送 THRESHOLD 块请求

« R1, R2 » C1 « A1, R3, R4, » C2, C3 « A3, R5, R6, » C4, C5, « A5, R7, R8, » C6, C7, » A7, R9, R10,

« R1, R2 » C1, « A1, R3, R4 » C2, « A2, R5, R6, » C3, « A3, R7, R8, » C4, C5, C6, C7, C8 ————– « A4, A5, A6, A7, A8

5.2 文件信息管理

5.3 网络驱动

5.4 系统集成

第 6 章

用户界面库

本章第一节先介绍一些造初期考虑使用的用户界面程序库 (GUI); 第二小节阐述了 HTML 作为本次设计的选择的原因以及使用这种技术会遇到的一些问题以及解决方案。第三小节简要的介绍了作为本次设计而附带产生的一款简单的 HttpServer。第四小节根据实际情况介绍了使用脚本语言来帮助我们简化开发。

6.1 常见 GUI 库

Win 平台专属的库微软本身的 MFC, Windows Forms[1], Borland 的 OWL(Object Windows Library); Mac 平台专属的库 Cocoa, Linux 平台由于操作系统本身是没有定义[2] GUI 的所以一般来说没有专属 GUI 库。由于多平台运行是本次设计的一个硬性目标所以以上 GUI 库无法采用。

能够提供跨平台支持的 GUI 库也是非常多, 但有的 GUI 库只提供一种 [3]编程语言如 Java 的 AWT, SWT, 所以这类 GUI 也不在考虑范围之类。

还有一些出名的的 GUI 库如 QT, GTK 虽然比较熟悉但这类 GUI 由于不仅仅包含了 GUI 的内容而且几乎是从底层完全重写导致编译出来的程序过大 [4], 因此考虑到最终程序的尺寸 (footprint) 问题, 犹豫之后还是放弃了这 2 个选择。

和 QT, GTK 类似的一款 wxWidget 在 windows 上优化之后可以让最终程序保持在 2M 左右, 也是我最熟悉的 GUI 库。

以上介绍的一些 GUI 库除了 Window Forms, AWT, SWT 因为和自身平台高度集成其外, 其他 GUI 库都由于 GUI 本身的复杂性以及历史原因导致有非常多的东西与

[1]dotnet 中的 winform

[2]unix 类系统一般使用 X11 作为 GUI。

[3]一般专门的 GUI 库都会有多种语言绑定

[4]根据以往经验使用 strip, upx 后也会达到 5M 左右, GTK 稍小。

现今标准中的内容重复,但又由于向后兼容以及这些内容已经在库中根深蒂固无法去除。

所以在初期还有考虑过 FLTK 这款简单的仅仅提供界面部分的 GUI 库。

但犹豫再三后还是决定放弃以上 GUI 库,主要原因在于

- 使用系统平台提供的 GUI 虽然可以不明显增加最终程序尺寸,但却不具备跨平台的特性;
- 使用平台无关的 GUI 一般会导致最终程序尺寸急剧膨胀;
- 一些小巧的 GUI 库虽然不会导致最终程序尺寸过大但此类 GUI 一般都较为简陋;

因此萌生使用 HTML 作为程序界面的想法,理由如下:

- HTML 依托用户的游览器本身来渲染界面,可以以最小的代价[5]来完成界面显示。
- 使用 JS 可以方便的处理界面逻辑,使用 CSS 灵活的定义显示界面。
- HTML 作为前端界面已经在暗中普及起来,只有稍微注意就可以发现仅仅国内,各大软件的最新版本中 Web 技术都已经在界面显示上占据越来越大的作用。
- HTML 可以使设计变得更简单,使用 WEB 技术意味着你在设计程序的时候就已经强制设计者是核心逻辑与界面逻辑分开处理。这样可以让强迫设计者时刻提醒自己核心逻辑应该提供出来的接口。同时在多人合作的时候可以并行展开工作。界面部分与核心部分可以做到没有任何耦合,他们直接的通讯是通过中间一层简单的包装进行联系的。

因此在选择 WEB 技术之后,可以满足程序尺寸的要求,并且能提供更加简单且灵活的界面。

6.2 选择 WEB 的问题

虽然使用 WEB 技术有非常多诱人的地方,但凡事都不会完美,WEB 也有着宁人烦恼的问题。

[5]程序大小方面。

- 由于种种原因在国内至今存在着大量已经被其开发者淘汰的游览器,这些游览器导致同一份文件最终显示效果却不同。
- 这一点和上面是对立的,就是如果在程序中内嵌自己的渲染部分则可以保证用户体验上的统一,但会急剧增加尺寸。
- 由于 WEB 技术和 C++ 或其他语言写的程序是独立的因此需要自己处理他们之间的通讯。
- 需要依赖服务器来提供内容。
- HTML/CSS/JS/IMG 这类文件和应用程序单独分离在发行程序的时候会稍显复杂。

对于存在的古老游览器我选择的是稍激进的方法 –不予理会。虽然第一感觉是不理智,但新技术是需要时间和人的推进才会普及起来,这也是现在部分非商业网站选择明确放弃这些游览器的原因。这是很多新应用新网站都必须进行的一个困难选择。不过好在各大厂商都在积极的推进 HTML5 技术而且和 C++11 类似,在标准还没正式发布前大部分功能已经在实践中运用了。这一问题的另外一个缓解方案是使用第三方的 JS 库来处理游览器的差异,这一点将在下一章中介绍。

内嵌一个游览器来给用户统一的体验是非常好的选择,但由于尺寸原因导致本文设计的这类程序无法接受。如果是其他程序如自用,定制产品或大规模的程序都可以选择这种方案,这里一般可以选择 WebKit 作为内核进行移植或使用第三方库,如果选择使用 QT 已经原生支持 WebKit 了,其他库或语言一般也会有对应 binding。

Web 页面需要和应用程序进行信息交换,这一点可以通过 jsonp 等常规方案解决简单解决,如果在 HTML5 普及之后也可以使用 WebSocket 进行更加完善的处理。

最后两点可以通过一种方案一次完美解决就是自己写一个简单的服务器,通过源码级别集成到应用程序里,Web 页面需要的一些文件在调式完成后就编译进这个服务器内部。

这里选择使用自己编写一款 HttpServer 的原因主要在于没有找到简单的解决处理最后一点的方案,如果仅仅是内嵌一款普通 HTTP 服务器有很多优秀的开源项目可供选择如 lighthttpd。

6.3 AppWebServer 的介绍

AppWebServer 是为了完成本次设计特意另行编写的一款专门用来集成到其他程序里的小型 HttpServer，且提供调式和发行两种模式。调式模式下 AWS 通过读取指定目录下的文件进行工作，发行模式下 AWS 则不读取任何外部文件，直接将文件编译到程序内部。

使用方式十分简单，如下代码就建立了一个简单的 HTTP Server，可以读取 doc_root(默认) 下的文件进行工作。

```
#inlcude <AppWebServer.hpp>
int main()
{
        AWS::Server s;
        s.run();
}
```

另外也可以通过注册 RPC::Service 到 Server 里去，这样游览器使用 js 发送一个请求到 “/rpc。cgi” 上就可以调用 Service 提供的函数来获得所需功能。js 端发送消息的格式会在下一章中进行介绍。这里只讨论后端代码。注册 Service 也是很简单的，如以下代码注册一个提供文件系统信息的 service 到 server 里去。

```
#inlcude <AppWebServer.hpp>
AWS::Service& rpc_filesystem();
int main()
{
        AWS::Server s;
        AWS::RPCServer rpc;
        rpc.install_service(rpc_filesystem());
        s.set_rpc(rpc);
        s.run();
}
```

```
\\...
AWS::Service& rpc_filesystem()
{
        static AWS::Service filesystem;
        //service的名称
        filesystem.name = "filesystem";

        //service的方法列表,以下语法使用C++11提供的lambda语法来编写匿名函数
        //使用Uniform initialization语法来初始化service的方法列表。
        //使用auto语法来自动推导函数返回类型。
        filesystem.methods = {
        {
                //函数名称
                "list_file",
                //具体的函数代码
                [](const AWS::JSON& j){
                AWS::JSON result;
                if (j["path"].isString()) {
                        string p(j["path"].asString());
                        for (auto& n : list_file(p)) {
                                AWS::JSON node;
                                node["type"] = n.type;
                                node["path"] = n.path;
                                node["name"] = n.name;
                                result.append(node);
                        }
                }
                return result;
                }
        },
        {
```

```
42                  \\ 提供的其他函数。
43          }
44          };
45          return filesystem;
46  }
```

一个 service 由 service 名称和方法列表组成,一般一个 service 会提供多个函数。其中 list_file 是完成具体工作的函数,这个函数接受一个路径然后返回此路径下所有的文件信息。AWS::JSON 底层使用的是 jsoncpp 的 json::value,JSON 参见 [3]。所有 service 的函数都会返回一个 AWS::JSON 类型来表达返回结果,程序核心部分和界面部分就是根据这些 service 提供的接口来完成设计的,如需要制作一个提供文件选择的界面模块则可根据这里提供的 filesystem service 来实现,WebPage 通过传递一个路径 (最开始可以是。代表当前目录) 给 “filesystem” 服务的 “list_file” 函数就可以得到这个目录下的文件信息,然后就可以继续根据得到的信息进行下一步请求。

这个模块也是这次设计中实际碰到的问题,由于文件传输肯定涉及到文件目录,但当前游览器基于安全考虑都是不允许得到文件路径的,通过 “<input type=file>” 只能得到文件的名称而无法获得完整路径。这在普通 Web 应用上是完全可以理解的,而且已经在现今所以较新版本游览器中执行了。所以必须要自己实现遍历文件系统的功能,最终实现结果比当初遇到这个问题预想的要容易多的解决。仅使用 100 来行的 C++ 和 100 行的 JS 代码就完成了这个功能。

下一节介绍完成 AWS 发行模式使用到的技术。

6.4 使用脚本语言来生成资源

linux 下曾多次遇到程序里需要嵌入外部文件的问题,Win 下可以方便的通过资源文件来解决,如 win 下的 icon 一般都是作为资源文件放到 pe 格式文件里的,并且通过一定的 hack 手段可以实现动态更新 exe 里的资源文件 [6]。Linux 下我曾多次寻找对应技术但没有什么收获,因此一般都是遇到的时候用脚本语言来生成C++代码作为补救。以前几次是使用 Perl 来编写,不过因为比较简单写的也比较凌乱代码没有保留,这次又碰到这个问题因此使用 Python 来写了一个特意完成

[6]比如可以实现配置信息保存到 exe 文件内。

AWS 发现模式功能的简单脚本来生成“资源文件”。脚本默认遍历读取 doc_root 目录下的文件，并生成 content.hpp 和 content.cpp 文件。content.cpp 里包含了资源内容，content.hpp 提供_RC 函数的接口，通过使用这个函数其他代码就可以方便的得到对应文件的字节码。如获得 index.htm 文件和 js/main.js 文件就可以简单的分别调用_RC(“/index.htm”) 和_RC(“/js/main.js”) 带得到，其中_RC 返回的是一个 array<const char*, size_t>，array 是 C++ 最新标版本 [6] 提供的标准类型。第一个字段指向对应文件的字节码，后一个字段表示文件长度。

脚本文件较为简单不到 100 行，主要是使用 struct 的 unpack 函数 [14] 来进行解码。代码如下

```
import sys
import os
from struct import unpack

content_hpp =  '\
/*THIS FILE IS AUTO GENERATE BY AppWebServer Resource Builder*/\n\
\n\
#ifndef CONTENT_HPP_\n \
#define CONTENT_HPP_\n\
#include <utility>\n\
#include <string>\n\
std::pair<unsigned char*, size_t> _RC(std::string path);\n\
#endif\n'

content_cpp = '\
/*THIS FILE IS AUTO GENERATE BY AppWebServer Resource Builder*/\n\
\
#include "content.hpp"\n\
#include <string>\n\
using namespace std;\n\
typedef unsigned char uchar;\n\
```

```
22          std::pair<uchar*,size_t> _RC(std::string path){{\n\
23      static uchar* _rc_empty = 0;\n\
24      {0}\n\
25      return make_pair(_rc_empty, -1);\n\
26  }}'
27  content_static = 'static uchar {0}[] = {{\n {1} \n}};\n'
28  content_return_first = '\
29  if (path == "{0}") {{\n\
30      return make_pair({1}, {2});\n\
31  }}'
32
33  content_return = '\
34   else if (path == "{0}") {{\n\
35     return make_pair({1}, {2});\n\
36  }}\n'
37
38  contents = ["", ""]
39
40  doc_root = "doc_root/"
41  if len(sys.argv) == 2:
42      doc_root = sys.argv[1]
43
44  def generate_one(name, path, template=content_return):
45      f = open(path, 'r')
46      data = f.read()
47      f.close()
48      if (len(data) == 0):
49          print "Waring: {0} is zero length file!".format(path)
50          c = template.format(path, "_rc_empty", 0)
51          return ["", c]
52
```

```
53      content = unpack("{0}c".format(len(data)), data)
54      tmp = ""
55      counter = 0
56      for i in content:
57          tmp += "0x{0:x}, ".format(ord(i))
58          counter += 1
59          if (counter % 10 == 0):
60              tmp += "\n\t\t"
61      s = content_static.format(name, tmp)
62      c = template.format(path[1:], name, len(content))
63      return [s, c]
64
65
66  old_path= os.path.abspath('.')
67  os.chdir(doc_root)
68
69  counter = 0;
70  for root, dirs, files in os.walk("."):
71      for file in files:
72          if (counter==0):
73              c = generate_one("_rc_{0}".format(counter),
74                               os.path.join(root, file), content_return_first)
75          else:
76              c = generate_one("_rc_{0}".format(counter),
77                               os.path.join(root, file))
78          contents[0] += c[0]
79          contents[1] += c[1]
80          counter += 1;
81
82  os.chdir(old_path)
83
```

```
84 f = open("content.cpp", "w")
85 f.write(content_cpp.format(contents[0]+contents[1]));
86 f.close()
87
88 f = open("content.hpp", "w")
89 f.write(content_hpp)
90 f.close()
```

这里的解决方案是可以算作是一种很早就有的使用用代码生成代码的方式，QT 的特定扩展，Google 的 Buffer Protocol 都算是这类解决方案。可以利用其他简单的工具辅助我们的工作。

AWS 通过这个脚本生成的 content。hpp 和 content。cpp 文件就能通过条件编译来选择是使用这里提供的数据还是直接读取文件系统下的目录。这样设计之后在代码调式的时候可以直接更改 WebPage 的内容，而在最终发布的时候便可以仅仅给用户提供一个可执行文件。

第 7 章

界面实现

WebPage 一般由三种文件组成:

1. HTML 即超文本标记语言。所以作为一种标记语言它提供的功能应该仅仅和 Markdown, textile 一样仅仅提供标记来让作者表达思想。HTML 通过 h1、p、div、ul、ol、table 等带有语义的标签来表达内容。HTML 本身和其他标记语言一样都是非常简单的,但用好却是需要经验的,现在依然有人乱用 table 标签作页面布局就是一个例子。[1] 如果是网站应该做到即使用户禁用 CSS/JS,用户依然可以获得主要信息,这也是 HTML 应该做的,使用一门标记语言就应该全面了解他每种标记的语义,不要乱用 [9]。

2. CSS 即层叠样式表,主要用来对页面进行布局以及对页面元素进行精确的外观控制。使用 CSS 后就可以根据一份内容生成多种外观[2],便于管理和设计。

3. JavaScript 是用来完成页面逻辑的[3]。它可以将一些以前需要在服务端完成的功能拿到游览器上来完成,减轻了服务器负担。而且随着 HTML5 的逐渐普及,使用 JS 可以完成以前只有桌面程序才拥有的功能[4]。

我们这里由于是使用 Web 技术来做本地应用程序而不是制作网站,而对于应用程序来说主要不是提供内容,而是进行数据处理的,所以会主要依赖 JavaScript 来完成逻辑部分和 CSS 来进行美化页面。接下来的两节会分别讨论他们。

[1] 由于 CSS 96 年才开始贝游览器支持,不过也已经出现 14 年了。
[2] 可以实现主题切换;针对其他设备访问比如手机,平板进行特定优化。
[3] 当然它也有进军服务端的趋势而且势头不小,比如 nodejs 已经类似于一门通用脚本语言了。
[4] 如 Cavans, Local Storage, WebGL, WebSocket 等。

7.1 更方便的 CSS–LeSS

在使用 CSS[12] 的时候会经常遇到一些重复定义的参数,这种情况写起来还好,但若遇上需要修改的时候就要一个一个找[5]。

还有如果你在遇到编写某个层次较深的 class 时,一长串前导符号写一次也没什么,但如果会有多个 class 需要定义则写着这些重复的字符串会让人觉得对不起自己。

有时候我们会定义某种主色 (或长度什么的),然后其他元素的颜色根据这个主色进行一定的调整,对于心算不行的朋友只能拿着一支笔或者开个计算器去算一下这可以接受但如果参考的值变动了,去修改那些依赖元素的时候又要小心查找修改了。

LeSS[13] 提供了变量,混合 (类似于继承),运算以及方便的 Color 函数,可以使用命名空间有组织的管理你的 css 文件。

学习 LeSS 是值得的,LeSS 只在 CSS 的基础之上增加了一些方便的语法,最终生成的还是普通 CSS 文件[6]。

7.2 写的更少做的更多 — JQuery

jQuery[5] 作为一款快速简洁的 JavaScript 库,它可以方便的操作 DOM,event,ajax 等。并且可以一定程度上减少各个游览器的差异。

在编写前端界面的时候,我们大量使用了 jQuery。

7.3 WebPage 与 C++ 通讯

首先引进一个 js 函数 rpc,他有 5 个参数依次是服务名 service; 方法名 method; 数据 data; 回调函数 function; 在通讯期间是否显示 notify。前 3 个参数对应于第 6.3节设计的 AWS::Service,回调函数是在异步通讯完毕后需要执行的函数,最后的 n 参数是可选的如果出现则在进行异步通讯到完成期间会出现一个滚动条之类的东西以便让用户知道发出的命令还在执行中,主要用在执行时间较长的调用中。

[5]如果你熟悉一种文本编辑器则会大大减轻这个时候的负担。

[6]可以直接在游览器通过 less。js 来生成最终的 css 文件或者在发布之前就使用一个 nodejs 程序 lessc 来生成 css 文件。

```
function rpc(s, m, d, f, n) {
        if (n) $("#ajaxnotification").show();
        $.getJSON("/rpc.cgi",
                encodeURI(JSON.stringify({service: s,  method: m,
                                        params: d, id: 0})),
                function(data) {
                        if (n) $("#ajaxnotification").hide();
                        if (data.error === null) f(data.result);
        });
}
```

其中 encodeURI 是必须的否则 IE 游览器会将 UTF8 的中文字符丢弃[7]导致通讯错误。另外这里只是简单的忽略了错误没有使用错误处理。

通过这个函数我们就可以方便的与 C++ 进行通讯,例如显示文件列表的核心函数 refresh_list。

```
refresh_list: function() {
        rpc("filemanager", "file_list", null, function(data){
                data = data || [];
                $("#c_file_list ul.list").html("");
                        var i;
                        for (i=0; i<data.length; i++) {
                                FileList.add_info(data[i]);
                        }
        });
}
```

这个函数通过使用 rpc 调用 C++ 中的 filemanager 服务的 file_list 方法,获得当前拥有的文件然后通过 FileList 对象的 add_info 函数来进行绘制这些文件到文件列表中。

[7]在 cp936 模式下。

因此在设计界面的时候几乎可以不用考虑数据问题,需要什么数据就直接通过 rpc 调用 C++ 提供的服务来获得数据。

第 8 章

开发中的测试

本章未完成

8.1 BoostPython 介绍

8.2 BoostTest 和 CPPUnit

结束语

本章未完成

表 8.1　Mac OS X 和 Windows 比较, 表格测试

项目	Windows	Mac
版本	Windows7	Mac OS x 10.6
核心	NT Kernel	XNU
CLI	CMD.exe	UNIX shell
GUI	Windows Explorer	Aqua

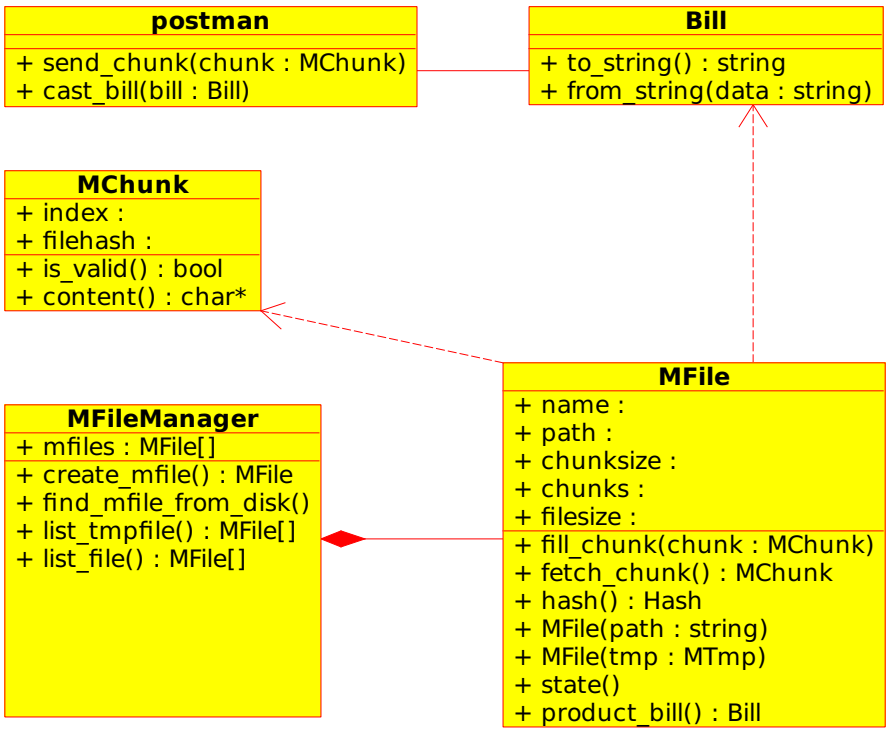

图 8.1　文件管理示意图, 图测试

参考文献

[1] RFC 768, User Datagram Protocol[S].

[2] RFC 793, Transmission Control Protocol[S].

[3] RFC 4627, The application/json Media Type for JavaScript Object Notation (JSON)[S].

[4] Andrej Cedilnik, HOWTO: Cross-Platform Software Development Using CMake ,2003-10

[5] jQuery 社区专家, jQuery Cookbook, 东南大学出版社, 2010-10

[6] Bjarne Stroustrup, C++11 -the recently approved new ISO C++ standard, http://www2.research.att.com/~bs/C++0xFAQ.html

[7] W.Richard Stevens, TCP/IP 详解卷 1, 机械工业出版社, 2000-4

[8] Bjarne Stroustrup, C++ 程序设计语言, 机械工业出版社, 2002-1

[9] Zoe Mickley Gillenwater, 灵活 Web 设计, 机械工业出版社, 2009

[10] Wikipedia, Hash Function, http://en.wikipedia.org/wiki/Hash_function

[11] Steve Oualline, Vi iMproved(VIM), Sams, 2001-4

[12] Christopher Schmitt, CSS Cookbook, O'Reilly Media, 2004-8

[13] Alexis Sellier, The Dynamic Stylesheet language, http://www.lesscss.net

[14] Mark Lutz, Python 编程 (第三版英文 · 影印版), 东南大学出版社, 2006

[15] Scott Chacon, Pro Git, Apress, 2009-8

[16] Boost Community, Boost Library Documentation, http://www.boost.org/doc/libs