小组会记录
20071024
author: pb

参加：yzf,pb,fankai,tqc,tsunami,zhulei

1。上周工作情况：

fankai:	
1. 完成测试计划 1 OK 2. 考虑 crash 情况下的测试计划	yzf：去除 exists, isdir 接口 tqc：修改 append 时，实现一个 chunk 分多次 buffer 的 append 调用
tqc:	
写 chunkserver 端的工作/功能列表 ：workload ：deleteChunk (与 master 通信) OK ：pipeline 目前 read 然后 write，能够重叠，可能是以后性能测试的一个地方 ：客户端 write 后要等待一会才能 read，why? ：master 重起后，chunkserver 必须要重起吗？	fankai：修改 datatransfer 接口，addChunk 不需要 chunkSize,
zhulei:	
完成 test_append 测试 OK	考虑加<uniqeid, dataMD5>做记录前缀，便于数据检查 OK
tsunami:	
完善 terminal 学习 FUSE OK	有兴趣的同学一起看 FUSE，看能否快速 demo 一下
yzf:	
在 paradise 实现基础上 考虑 Mapred 调度部分的设计 OK	tqc：master 重起后，chunkserver 必须要重起吗？ fankai：去除 exists, isdir 接口
pb:	
辅助考虑 Mapred 的设计	BUG：file size 没有正确记录 OK BUG：root ////的情况处理 OK

2。本周工作安排

fankai:	
完成 crash 情况下的测试计划	Client Lib 缺省参数 Client Lib 的注释/文档，使用指南
tqc:	
完成 workload 功能	1. 写 chunkserver 端的工作/功能列表 2. fankai ：修改 datatransfer 接口，addChunk 不需要 chunkSize, 3. 客户端 write 后要等待一会才能 read，why? 4. master 重起后，chunkserver 必须要重起吗？
zhulei:	
完成 test_append 测试	辅助 fankai 一起进行 crash 情况的测试计划
tsunami:	
terminal 键盘 history,快捷键支持	1. shell 用户友好性，参考 lftp 实现 2. BUGFIX: append 大文件的问题
yzf:	
Mapred 调度部分的设计(在 paradise 实现基础上)	tqc：master 重起后，chunkserver 必须要重起吗？
pb:	
进行 UnitTest 和 CodeReview 的工作	辅助 yzf 考虑 Mapred 的设计

3。Presentation

Pragmatic Unit Testing

The Pragmatic Programmers

Pragmatic Unit Testing

In C# with NUnit

The Pragmatic Starter Kit – Volume II

Andrew Hunt David Thomas

The Pragmatic Programmers

Pragmatic Unit Testing

In Java with JUnit

The Pragmatic Starter Kit – Volume II

Andrew Hunt David Thomas

Coding With Confidence

- A *unit test* is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested.
- All we want to do is prove that code does what we intended, and so we want to test very small, very isolated pieces of functionality.
- After all, if we aren't sure the code is doing what we think, then any other forms of testing may just be a waste of time.

What Do I Want to Accomplish?

- Does It Do What I Want?
  - You want the code to prove to you that it's doing exactly what you think it should.
- Does It Do What I Want All of the Time?
- Can I Depend On It?
- Does it Document my Intent?
  - executable documentation has the benefit of being correct.

Excuses For Not Testing

- It takes too much time to write the tests
  - isolating a reported bug
  - debugging code
  - reworking code that you thought was working
- I don't really know how the code is supposed to behave so I can't test it
  - If you truly don't know how the code is supposed to behave, then maybe this isn't the time to be writing it.
- My company won't let me run unit tests on the live system
  - While you might be able to run those same tests in other contexts (on the live, production system, for instance) *they are no longer unit tests*.

What to Test?

- Right — Are the results right?
- B — Are all the boundary conditions CORRECT?
- I — Can you check inverse relationships?
- C —Can you cross-check results using other means?
- E — Can you force error conditions to happen?
- P —Are performance characteristics within bounds?

Are the results right?

- *If the code ran correctly, how would I know?*

possible boundary conditions

- Conformance
  - Does the value conform to an expected format?
- Ordering
  - Is the set of values ordered or unordered as appropriate?
- Range
  - Is the value within reasonable minimum and maximum values?
- Reference
  - Does the code reference anything external that isn't under direct control of the code itself? What other conditions must exist in order for the method to work?
- Existence .
  - Does the value exist (e.g., is non-null, nonzero, present in a set, etc.)?
- Cardinality
  - Are there exactly enough values?
- Time (absolute and relative)
  - Is everything happening in order? At the right time? In time?

Using Mock Objects

- A mock object is simply a debug replacement for a real world object.
  - The real object has nondeterministic behavior (it produces unpredictable results, like a stock-market quote feed.)
  - The real object is difficult to set up.
  - The real object has behavior that is hard to trigger (for example, a network error).
  - The real object is slow.
  - The real object has (or is) a user interface.
  - The test needs to ask the real object about how it was used (for example, a test might need to confirm that a callback function was actually called).
  - The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

How to mock in XUnit?

- 1. Use an interface to describe the object
- 2. Implement the interface for production code
- 3. Implement the interface in a mock object for testing

Mock in CxxTest

Properties of Good Tests

- Automatic
- Thorough
  - you may want to invest in code coverage tools to help
- Independent
  - every test should be independent from every other test
  - only testing one thing at a time, concentrate on a single production method
- Repeatable
  - they must be independent of the environment as well.
- Professional
  - be written and maintained to the same professional standards as your production code.

How to Fix a Bug

- 1. Identify the bug.
- 2. Write a test that fails, to prove the the bug exists.
- 3. Fix the code such that the test now passes.
- 4. Verify that ***all*** tests still pass (i.e., you didn't break anything else as a result of the fix).

Could this same kind of problem happen anywhere else?

Test on the Project

- Test Courtesy
  - all tests pass all the time.
  - When code commit, it has complete unit tests, and that is passes all of them.
- Test Frequency
  - Write a new method
  - Fix a bug →*verification testing* , *regression testing*
  - Any successful compile
  - Each check-in to version control
  - Continuously
- Tests and Reviews
  - Reviews of the test code are incredibly useful.
  - Give a check list on review

Designing for Testability

```
public void SleepUntilNextHour() {
  int howlong;
  // Calculate how long to wait...
  Thread.Sleep(howlong);
  return;
}
```

```
public void SleepUntilNextHour() {
  int howlong = MilliSecondsToNextHour(DateTime.Now);
  Thread.Sleep(howlong);
  return;
}
```

General Principles:

- ☐ Test anything that might break
- ☐ Test everything that does break
- ☐ New code is guilty until proven innocent
- ☐ Write at least as much test code as production code
- ☐ Run local tests with each compile
- ☐ Run all tests before check-in to repository

Questions to Ask:

- ☐ If the code ran correctly, how would I know?
- ☐ How am I going to test this?
- ☐ What *else* can go wrong?
- ☐ Could this same kind of problem happen anywhere else?

What to Test: Use Your RIGHT-BICEP

- ☐ Are the results **right**?
- ☐ Are all the **boundary** conditions CORRECT?
- ☐ Can you check **inverse** relationships?
- ☐ Can you **cross-check** results using other means?
- ☐ Can you force **error conditions** to happen?
- ☐ Are **performance** characteristics within bounds?

Good tests are A TRIP

- ☐ **A**utomatic
- ☐ **T**horough
- ☐ **R**epeatable
- ☐ **I**ndependent
- ☐ **P**rofessional

CORRECT Boundary Conditions

- ☐ **C**onformance — Does the value conform to an expected format?
- ☐ **O**rdering — Is the set of values ordered or unordered as appropriate?
- ☐ **R**ange — Is the value within reasonable minimum and maximum values?
- ☐ **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- ☐ **E**xistence — Does the value exist? (e.g., is non-null, non-zero, present in a set, etc.)
- ☐ **C**ardinality — Are there exactly enough values?
- ☐ **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?